CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Division
Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

54021

0-315-06101-4.

# PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

- Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

DOUGLAS J. RIDEOUT

Date of Birth — Date de naissance: MAY 6, 1955

Country of Birth — Lieu de naissance: CANADA

Permanent Address — Résidence fixe

48 CHESTNUT ST.
YARMOUTH, NOVA SCOTIA B5A 2N6

Title of Thesis — Titre de la thèse

AN APPLICATION OF A MICROCODE COMPACTION TECHNIQUE TO THE NANODATA QM-1 NANO-ARCHITECTURE

University — Université

UNIVERSITY OF ALBERTA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF SCIENCE

Year this degree conferred — Année d'obtention de ce grade: 1981

Name of Supervisor — Nom du directeur de thèse: DR. SUBRATA DASGUPTA

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date: AUG. 21, 1981

Signature: DJ Rideout

NL-91 (4/77)

THE UNIVERSITY OF ALBERTA

# AN APPLICATION OF A MICROCODE COMPACTION TECHNIQUE TO THE NANODATA QM-1 NANO-ARCHITECTURE

by

© Douglas J. Rideout

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF Master of Science
IN
Computing Science

Department of Computing Science

EDMONTON, ALBERTA
Fall, 1981

THE UNIVERSITY OF ALBERTA

**RELEASE FORM**

NAME OF AUTHOR Douglas J. Rideout

TITLE OF THESIS AN APPLICATION OF A MICROCODE COMPACTION TECHNIQUE TO THE NANODATA QM-1 NANO-ARCHITECTURE

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall, 1981

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) Douglas J. Rideout

PERMANENT ADDRESS:

48 Chestnut Street
Yarmouth, Nova Scotia
B5A 2N6

DATED August 21 1981

THE UNIVERSITY OF ALBERTA
FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled AN APPLICATION OF A MICROCODE COMPACTION TECHNIQUE TO THE NANODATA QM-1 NANO-ARCHITECTURE submitted by Douglas J. Rideout in partial fulfilment of the requirements for the degree of Master of Science in Computing Science.

Supervisor

Pat Harding

Stan Sutphen

Date August 21, 1981

# Abstract

Microcode compaction is the process of detecting potential concurrencies between microoperations, and placing them in microinstructions so as to take maximum advantage of such parallelism. Local compaction is the task of compacting microoperations within a segment of code having only one entry point and one exit point at the end, and no branches in between. The highly horizontal and complex organization of the lower level control store (the nanostore) of the Nanodata QM-1 provides an ideal testing ground for a "real world" evaluation of the theory of local microcode compaction. Further, determining the feasibility of compaction of QM-1 nanocode is an important step in the development of a high-level language for nanoprogramming the QM-1.

The theory of microcode compaction provides a general model of machine behavior, and definitions of algorithms for various approaches to the problem. The model is shown to be general enough to apply to the QM-1, with a few minor variations. The algorithm used was found to be defined well enough to allow easy implementation of its general flow.

This study shows that compaction of QM-1 nanocode is possible, and the implemented compactor produces code that compares well with hand-generated code.

# Acknowledgements

I would like to thank my supervisor, Dr. Subrata Dasgupta, for all of his help throughout the course of this work. His support and encouragement were inspiring, his enthusiasm was infectious, and his many suggestions and criticisms were invaluable.

I am also indebted to the other members of the "QM-1 Group", namely Steven Sutphen, Marius Olafsson, and Alynn Klassen, with whom I spent many hours in most fruitful discussions. Their interest in related topics and the ideas that came from our discussions were of great benefit to this work.

Further thanks are due to Alynn Klassen for preparing some of the Figures. Thanks also to Donna Fremont and Dr. Anne Brindle for carefully reading a draft of this thesis, and for their many helpful comments and suggestions.

Finally, I am deeply grateful to my wife, Barbara, who helped with this work in many ways, but whose contribution in terms of her support, encouragement, patience, and understanding is immeasurable. I cannot begin to thank her enough.

## Table of Contents

Chapter Page

1. Introduction ........ 1

   1.1 Organization of the Thesis ........ 3

2. Microcode Compaction--the Problem ........ 4

   2.1 Microinstruction Format ........ 4

   2.2 Compaction ........ 4

   2.3 Local vs. Global Compaction ........ 5

   2.4 Compaction vs. Optimization ........ 6

   2.5 Complexity of the Problem ........ 7

3. Compaction Methodology ........ 8

   3.1 MO Representation ........ 8

   3.2 Analyses of MO Interaction ........ 9

      3.2.1 Data Dependence ........ 9

      3.2.2 Resource Conflicts ........ 11

   3.3 Extensions of the Machine Model ........ 11

      3.3.1 Resource Allocations ........ 12

      3.3.2 Input/Output Time Constraints ........ 12

      3.3.3 Non-static Storage Resources ........ 14

   3.4 The Algorithms ........ 14

      3.4.1 Linear Analysis ........ 15

      3.4.2 Critical Path ........ 16

      3.4.3 Branch and Bound ........ 17

         3.4.3.1 BAB Heuristics ........ 18

   3.5 Implementation ........ 19

4. Impact of the QM-1 Structure on Compaction ........ 20

   4.1 Nanoword Structure ........ 20

   4.2 Use of K-fields ........ 23

   4.3 Residual Control ........ 23

   4.4 Other Indirect Specification of Resource Usage ........ 26

   4.5 Local Store Register 31 ........ 28

4.6 Timing Considerations 29
   4.6.1 Non-Executable Operations 30
   4.6.2 Leading Edge Operations 31
   4.6.3 Trailing Edge Operations 31
   4.6.4 Delay Operations 32
5. Implementation of the Machine Model 34
5.1 The Tuple 34
5.2 Extensions 37
   5.2.1 Resource Binding 37
   5.2.2 Nanooperation Versions 38
   5.2.3 Time Constraints 39
   5.2.4 Delays 40
5.3 Volatile Resources and Bundling 41
5.4 Analyses of Nanooperation Interaction 44
5.5 Summary 45
6. The Linear Algorithm 46
6.1 Approach of the Algorithm 47
6.2 Version Shuffling 48
6.3 Operation of the Algorithm 49
6.4 S* Special Constructs 51
7. The S*(QM-1) Compiler-Compactor Interface 52
7.1 Intermediate Language 52
7.2 Control Operators 53
7.3 Sequencing Considerations 54
   7.3.1 Sequencing Within SLMs 54
   7.3.2 Sequencing from the End of an SLM 55
7.4 S* Special Constructs 58
8. The Implementation 59
8.1 The Environment 59
8.2 Strengths and Weaknesses 59
8.3 Delay NO Considerations 60

8.4 Compaction Examples ........ 61
8.5 Analysis of Compactor Performance ........ 61
9. Conclusions ........ 65
9.1 Applicability of the General Model ........ 65
9.2 Time and Space Minimization ........ 66
9.3 Machine Organization ........ 67
9.3.1 Using Empty Instructions ........ 69
9.3.2 Higher Level Concurrency Specifiers ........ 69
9.4 Using S* Constructs ........ 72
9.5 Size of SLMs ........ 73
9.6 Time Complexity of the Implemented Compactor ........ 74
9.7 Topics for Further Study ........ 74
Bibliography ........ 76
APPENDIX I ........ 82
APPENDIX II ........ 85
APPENDIX III ........ 90
APPENDIX IV ........ 96

# List of Figures

Figure Page

1. S*(QM-1) Compilation Process 3
2. Example of a DDG 10
3. Nanoword Format 21
4. QM-1 Busing Structure 22
5. Residual Control of ALU Input and Output 24
6. Selection Hierarchy for INDEX ALU Left Input 27
7. QM-1 Timing 29
8. NO Table Representation of 'read cs(arg)' Operation 35
9. NO Sequence 42
10. NO Sequences Causing Bundle Exceptions 44
11. NO Sequence for an ALU Operation 49
12. Stages in Packing the NO Sequence 50
13. Nanoprogram Organization in Nanostore 55
14. 'goto' Statement Sequencing Conventions 57

15a. Loop in S*(QM-1) Code 67

15b. Loop with Residual Control Inside Loop Nanoword 67

15c. Loop with Residual Control Outside Loop Nanoword 68

16. Sequence Construct 70
17. Parallel Construct 70
18. SLM Without Cocycle 72
19. SLM With Cocycle 72

# Chapter 1

# Introduction

In recent years, many studies have been done on the problem of microcode compaction [YAU74, DASG77, TOKO77, MALL78]. This refers to the task of detecting concurrency between microoperations expressed in a sequential list, and placing them in microinstructions to take maximum advantage of the parallelism of the machine. The theory of "local compaction", in particular, has been well developed and surveyed of late [LAND80]. Local compaction is the task of packing into microinstructions a sequence of microoperations having only one entry point and at most one branch (at the end of the sequence), such that if one operation in the sequence is executed, then all are executed.

There has been a large amount of interest in development of high-level microprogramming languages [DASG74, ECKH71, PATT76, SALO76, DASG78a] in order to reduce the difficulty of microprogramming. The two problems are very closely related in that the usefulness of any high-level microprogramming language (HLML) depends heavily on the efficiency of the object microcode generated [DASG76b]. The efficiency of the code is determined by the extent to which the parallelism possible between microoperations is exploited.

The methods of compaction, and the models upon which the methods are based, must now be subjected to stringent tests by application to "real world" microprogrammable machines. Implementations of microcode compaction methods have previously been undertaken [MALL78, FISH79], but it is not clear from the literature how *realistic* the code being compacted was in terms of degree of parallelism possible, timing considerations, data interactions among operations, and type of hardware control.

An application of a local microcode compaction technique to the Nanodata QM-1, a microprogrammable machine, is presented in this work. The complexities of the architecture of this machine, whose organization at the lowest level of control store exhibits a high potential for concurrency of operations, provides a particularly difficult (and therefore important) testing ground for compaction theory.

In addition to testing the theory of compaction in general, the present study serves as an experiment into determining the feasibility of compacting QM-1 nanocode[1]. "Nanoprogramming" the QM-1 is at best a tedious task due to the structure of its nanoword, the residual control exercised over hardware resources, complex timing requirements, and the high degree of parallelism possible. (All of these features are further discussed in later chapters.) Automatic compaction of nanocode is one step toward making the use of the QM-1, at the nano-level, easier.

Microprogramming is intimately tied to machine architecture, and thus one cannot expect an HLML to be completely machine-independent. On the other hand, if an HLML is ever to be useful for microprogramming on a variety of machines having different architectures, then there *must* exist that same machine-independence. In response to these opposing requirements, Dasgupta has proposed a microprogramming language schema, called S* [DASG78a, DASG80a, DASG80b]; this is, in fact, a language whose semantics are only partially defined. Constructs which may be considered machine-independent are provided in the partial language definition which is S*. Machine-specific details are filled in when the schema is "instantiated" into a language, S*(M1), for a particular machine M1, on the basis of the structure of that machine.

As a test of the feasibility of this schema concept, Klassen has undertaken [KLAS81a] the instantiation of S* for the Nanodata QM-1 [NANO72], and the design of a compiler for S*(QM-1).

Given that an HLML, specifically S*(QM-1), is to be developed and a compiler implemented, microcode compaction must be made available in order that the QM-1 nanocode produced from S*(QM-1) will be efficient to execute. A further consideration for this machine is that a nanoprogram be small enough to reside in the limited nanostore, which can contain a maximum of 1024 words.

Local compaction serves as one of the final stages in the process of translating S*(QM-1) source code to executable nanowords for the QM-1 (Fig. 1). The compactor is designed to accept and pack code from sources other than the S*(QM-1) compiler, such as code compiled from other HLMLs, or hand-generated sequential code, as long as the input conforms to certain conventions established to provide the compactor with

---

[1] The QM-1 possesses a two-level control store, the lower of which is called *nanostore*, and hence the code at that level is known as *nanocode*. Compaction is applied at the nano-level.

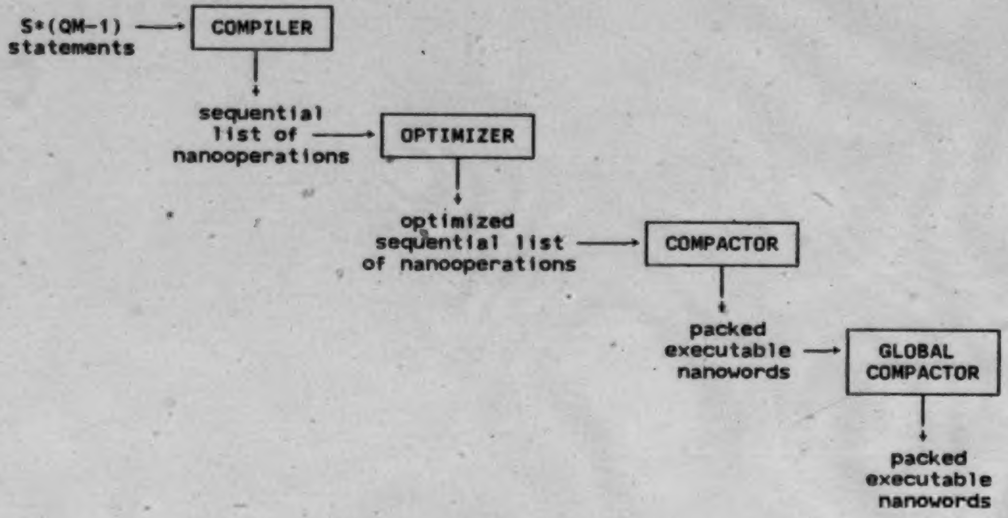

S*(QM-1) statements
COMPILER
sequential list of nanooperations
OPTIMIZER
optimized sequential list of nanooperations
COMPACTOR
packed executable nanowords
GLOBAL COMPACTOR
packed executable nanowords

Fig. 1. S*(QM-1) Compilation Process

sequencing and addressing information.

### 1.1 Organization of the Thesis

The next two chapters of this thesis present the problem of local microcode compaction, and the current approaches to the solution of the problem, in the context of a common model of machine behavior as described in the literature.

Chapter 4 describes some of the features of the QM-1 architecture and their impact on the compaction of nanocode. Chapters 5 and 6 outline how the practice matches the theory in QM-1 nanocode compaction; i.e. they contain discussions on how the implementation conforms to the machine model and the compaction algorithm.

Chapter 7 deals with the interface between the compactor and the S*(QM-1) compiler, and Chapter 8 presents some examples and an analysis of compactor performance.

Finally, the last chapter discusses the results of the work, and presents some suggestions for further study.

## Chapter 2

## Microcode Compaction--the Problem

### 2.1 Microinstruction Format

A microoperation (MO) is the most primitive machine activity that is considered here. (An MO could be the operation of gating the contents of a bus to a register, or that of reading the contents of a specified location in storage onto a data bus). A microinstruction (MI) may be characterized as a set of MOs, while a sequence of MIs makes up a microprogram.

Microinstructions are generally categorized as either *vertical* or *horizontal*, but there are no "universal" definitions of the terms [SALI76]. A vertical MI is usually considered to be one which is highly encoded, often containing only one MO, leading to a non-parallel organization [DASG79]; it thus resembles the more familiar assembler language instruction. The horizontal MI is typified by the ability to "exercise simultaneous control over all independent hardware facilities, with encoding limited to mutually exclusive control signals" [SALI76]. These organizations represent the opposing ends of the scale, and there exist varying degrees between the two; in fact, the dividing line is quite fuzzy. Word organizations having highly (or maximally) encoded MIs, leading to microword lengths on the order of 16 to 32 bits are also referred to as vertical, even though they may contain several MOs executing in parallel [DASG79], and thus horizontal versus vertical is often characterized by long versus short words.

### 2.2 Compaction

The microcode compaction problem has been (more or less) formally defined in [LAND80] in a manner similar to the following:

> For a particular machine, a given microprogram is expressed as a sequence of MOs which are to be placed into MIs in such a way as to minimize the execution time of the microprogram, under the restriction that the resulting

(compacted) sequence of MIs is "semantically equivalent" [2] to the original sequence of MOs.

Microcode compaction, therefore, amounts to the detection of concurrency between MOs; this involves maintaining the order of MOs that pass data to each other (data dependence analysis), and detecting when MOs require the same machine resource (resource conflict analysis). These are discussed in more detail in Chapter 3. It is clear that compaction is only useful and interesting for machines of horizontal organizations (i.e. the MI is horizontal), where MO concurrency is possible.

### 2.3 Local vs. Global Compaction

To facilitate compaction, a microprogram is usually divided into straight-line microcode segments (SLMs), also known as basic blocks. An SLM is "an ordered collection of MOs with no entry points, except at the beginning, and no branches, except possibly at the end." [LAND80].

Local compaction is the compaction of MOs within an SLM, where minimization of the number of MIs generally results in minimization of execution time [FISH79, MALL78]. Local compaction is applied (individually) to each SLM in the given microprogram.

Much active research is also underway on the subject of global compaction of microprograms [TOKO78, DASG79, WOOD79, FISH79, POE80], which involves compaction across SLMs. Since the length of an SLM tends to be short, global compaction generally has a greater effect on program performance than local compaction, although it is a more complex problem because minimization of the number of MIs does not necessarily minimize execution time.

Opinions vary on just how effective local compaction is when used in conjunction with a global compactor. Certainly, the shorter the SLMs, the less will be the impact of local compaction on the generation of efficient code. However, local compaction is generally recognized as important, whether it is seen as a "fundamental part of any compaction process" [LAND80], or as a basic approach to be extended to global analysis [TOKO78, FISH79].

---

[2] "Semantic equivalence" implies that both sequences always result in the same output for the same input [LAND80 ]; perhaps a better term would be "functionally equivalent."

The discussions on whether to compact microcode locally before globally, or to pack globally from the start, are still ongoing, as is the development of the theory of global compaction itself. Unless S*(QM-1) compiler output is compacted, by whatever approach, the code will be useless. This study is confined to the development of a local compaction routine for S*(QM-1) nanocode, recognizing the possibility, and probably the necessity, of adding global compaction when the studies have established the approach to that problem sufficiently.

### 2.4 Compaction vs. Optimization

The term "optimization" has been used in the literature to refer to the process that has been called "compaction" here; i.e. detection of parallelism among MOs. In fact, optimization has been used to refer to a whole range of operations from bit dimension reduction of control store words [DASG79, SRIM80, AGER76], to elimination of redundant operations [SITT73, DOMA75].

The term "microcode optimization" is taken here to have meaning as it has been viewed in some of the more recent literature [MALL78, FISH79]. Microcode optimization performs the same code transformations as does compiler optimization, such as the identification and deletion of nonessential actions, and constant folding [SITT73, LEE74].

Thus compaction and optimization are different tasks, and in fact, while optimization is applicable to vertical control store organizations, compaction is not. They are performed in different phases in order to "obtain high efficiency of processing time and memory requirement" [TOKO78]. It will be assumed here, then, that microprogram optimization has already been performed when local compaction is begun, although whether that is the case or not has little or no bearing on this study of the local compaction process.

## 2.5 Complexity of the Problem

The ultimate goal of microcode compaction is, of course, to obtain optimally compacted code; i.e. code which could not be placed into MIs in any alternate way resulting in the use of fewer MIs while maintaining functional equivalence. This problem, however, has been shown to be NP-Complete [LAND80][3], so that guaranteed optimal packing of an SLM seems to require time exponential in the number of MOs in the SLM. This is a consequence of the fact that, in order to ensure optimality of compaction, every legal[4] combination of placing MOs in MIs must be considered.

Hence, the goal of guaranteed optimal compaction in all cases is unattainable in time that justifies the compaction procedure. However, some very good heuristics exist for the various methods of compaction, and, while they do not *guarantee* a minimal packing, empirical tests done to date show that the resulting code approaches or reaches optimality in many cases, while the time complexity is reduced to be at most polynomial (and sometimes linear) in the number of MOs [TOKO78, WOOD78, MALL78, FISH79].

---

[3] See also [DEWI76] for a proof that includes resource allocation in the model, and [YAU74].

[4] Legal in the sense that data dependencies are preserved, and resource conflicts avoided; i.e. functional equivalence is maintained.

# Chapter 3

# Compaction Methodology

## 3.1 MO Representation

Discussion of any algorithm for microcode compaction must be undertaken within the framework of a model of machine behavior. This implies that it must be possible to specify the primitive activities (MOs) of a machine in a precise, complete, manageable and, where possible, machine-independent way. The model must include enough information to enable identification of interactions between operations. Most of the models used in the literature are somewhat similar [KLEI74, DASG76a, DEWI76, MALL78, RAMA74, YAU74], and a synthesis of these is presented in [MALL78] and [LAND80]. This is the model adopted here and discussed below.

In this model, the MO is represented as a six-tuple:

<name, I, O, U, T, F>

where the tuple elements are

name – an identifier of the MO to be performed;

I – set of all storage resources required by the MO as input;

O – set of all storage resources into which the MO places output;

U – set of all functional units required by the MO while executing;

T – set of processor clock phases required for MO execution;

F – set of all MI fields required by the MO.

It is nowhere claimed that the above is in any way the best representation for modeling machine behavior, but it at least satisfies the criteria for a machine model [LAND80]. Moreover, it has been used successfully in implementations [MALL78].

The tuple representation is clarified by means of an example wherein a QM-1 nanooperation[5] is displayed. An operation exists on the QM-1 for incrementing the microprogram counter (mpc), which is simply a register. The mpc may be incremented by any of several different values (including 1 and 2), and the actual operation is performed by a special unit called the mpc Increment Facility (*mpc incr fac*). There are four registers in the QM-1 that may act as the mpc, and the particular one of these that performs that function at any given time is designated by the special register *fmpc*.

---

[5] Only the necessary details are presented here. Descriptions of some of the features of the QM-1 structure, and the application of the model to the machine appear in later chapters.

The operation is encoded in a field of an instruction designated *im*, and the value by which the mpc is to be incremented is placed in another field called *g spec*. The operation is performed on the "trailing edge" (the end of the pulse) of the R-clock (one of the "phases" of the QM-1 clock). This timing information is designated by "RT".

The tuple representation of the increment mpc operation is

<inc mpc, {mpc, fmpc}, mpc, mpc incr fac, RT, {im, g spec}>.

The elements I, O, U, T, and F in the six-tuple are known as tuple sets, since I and O each may represent a set of storage resources used, U a set of functional units, etc. That notion is useful in data dependence and resource conflict analysis. For a given MO, say MOi, the tuple sets of MOi are denoted by Ii, Oi, Ui, Ti, and Fi, respectively.

## 3.2 Analyses of MO Interaction

### 3.2.1 Data Dependence

A data interaction exists between two MOs whenever one MO writes to a storage resource that another MO either reads or writes. More precisely, suppose that MOi precedes MOj in the given SLM; then MOj is **data dependent** on MOi (i.e. MOi dd MOj) if and only if any of the following hold:

1. An output resource of MOi is also an input resource of MOj; i.e. $Oi \cap Ij \neq \emptyset$
2. An input resource of MOi is also an output resource of MOj; i.e. $Ii \cap Oj \neq \emptyset$
3. An output resource of MOi is also an output resource of MOj; i.e. $Oi \cap Oj \neq \emptyset$

If at least one of these conditions holds, then by interchanging the order of MOi and MOj, either the wrong value is left in the common resource(s) (Case 3), or one of the two MOs receives an incorrect input value from the common resource(s) (Cases 1 and 2).

If two MOs are not data dependent, then they are said to be **data independent** (i.e. MOi di MOj).

Some compaction methods utilize a *data dependency graph* (DDG) to represent the partial ordering imposed on MOs in an SLM by data dependence. The DDG is a

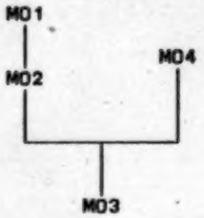


**Fig. 2. Example of a DDG**

directed graph possessing the properties that:

1. there exists a directed path in the graph from MOi to MOj, iff MOi precedes MOj in the SLM and MOi dd MOj;
2. there exist no redundant edges in the graph (i.e. no edge can exist from one MO to another if there is another directed path between them).

Figure 2 shows a DDG with MO1 dd MO2, MO2 dd MO3, MO4 dd MO3; no edge will exist between MO1 and MO3.

An edge between MOi and MOj in a DDG implies that the two MOs cannot coexist in the same MI because of their data dependence. If, however, as is often the case (and it is for the QM-1), MOs do not use the machine resources for the entire MI cycle time, then two microoperations may be executed at disjoint times in the microcycle. This describes the definition of timing precedence, which is stated more formally as:

> MOi **timing precedes** (tp) MOj if and only if $Ti \cap Tj = \emptyset$ and clock phase *a* precedes clock phase *b* for $a \in Ti$ and $b \in Tj$.

Two microoperations may coexist in a microinstruction, with respect to data dependence, if either they are data independent or they satisfy the timing precedence relation. So the following relation is defined:

> If MOi precedes MOj in an SLM, then MOi and MOj
> are **data compatible** (i.e. MOi dc MOj) iff MOi di MOj or MOi tp MOj.

The existence of **weak dependencies** between MOs (i.e. when MOj is data dependent on MOi but the two are data compatible) implies that not all MOs joined by edges in the DDG are forced into separate MIs.

### 3.2.2 Resource Conflicts

In addition to data dependence analysis, the compaction process must also establish when there are potential conflicts over resource usage between MOs; specifically, this analysis must recognize attempts to use functional units or microinstruction fields simultaneously.

In terms of tuple sets, then, two MOs, MOi and MOj, are considered to be **unit compatible** if and only if

$Ui \cap Uj = \emptyset$ or $Ti \cap Tj = \emptyset$.

That is, MOi and MOj are unit compatible if they do not attempt to use the same functional unit during the same clock phase. For example, a machine possessing only one ALU cannot perform ADD and SUBTRACT operations at the same time.

MOi and MOj are **field compatible** (also often referred to as **residence compatible**) if and only if

$Fi \cap Fj = \emptyset$ or $\forall$ fields f such that $f \in Fi$ and $f \in Fj$, the value to be placed in f by MOi is the same as the value to be placed in f by MOj.

The second part of the definition allows two MOs to use the same field of the MI simultaneously if they both place the same value in the common field.

It has been shown [MALL78] that if two operations MOi and MOj are data compatible, unit compatible, and field compatible, then they are **parallel** (i.e. they can reside in the same MI and can thus be executed within the same MI cycle time). Conversely, MOi **conflicts** with MOj (i.e. MOi c MOj) if MOi is not parallel to MOj.

### 3.3 Extensions of the Machine Model

The outline of the machine model presented in Section 3.1 is extended in [MALL78, LAND80] to allow for resource allocation, timing considerations, and non-static resources.

### 3.3.1 Resource Allocations

Tuple sets in the MO tuple representation are considered to be either **bound** or **unbound** with respect to machine resources allocated to them. An unbound set contains a list of all *possible* resources that can be allocated for use by the MO; a bound set contains a list of those resources actually allocated for use by the MO. For example, if MOi is capable of using either the ALU or the INDEX ALU of a machine, then the unbound U tuple set is

Ui = {ALU or INDEX ALU}.

The content of Ui above is known as an "Orlist" [LAND80]. When a choice is made as to which of the two units that MOi is to use (i.e. when the resource is bound to MOi), a **version** of the tuple set results:

UiVer1 = {ALU}

UiVer2 = {INDEX ALU}.

Tuple sets may also contain "Andlists". For binding, one element must be chosen from each Orlist grouped together by the Andlist. (The conjunction is usually denoted by a comma). For example, given the unbound U set:

Uj = {(ALU1 or ALU2), (ALU3 or ALU4)},

one version of Uj is

UjVer1 = {ALU1,ALU3}

which, as in the previous example, allocates resources to the MO in question.

These Orlist and Andlist semantics may be used in any of the tuple sets. Versions of an NO may be enumerated and stored as a group for use during the actual compaction.

### 3.3.2 Input/Output Time Constraints

Extensions to the model are presented in [MALL78] to take into consideration the cases where MOs use storage resources for only a subset of the clock phases listed in the T set for that MO, and where MOs require use of machine resources for more than one MI cycle time.

The first of these possibilities is best illustrated by an example. Suppose that MOi writes to register R2, but completes by the end of Phase1 of the microcycle, and that MOj begins in the first phase (Phase1) of the cycle but does not require R2 as an input until Phase2, which begins after the end of Phase1. Thus the relevant tuple sets in their original form would be:

$$Oi = \{R2\},\ Ti = \{Phase1\}$$

$$Ij = \{R2, R3\},\ Tj = \{Phase1, Phase2\}$$

These do not reflect the timing of resource usage that results in a weak dependence, so the required information is entered into the tuple sets in the following manner:

$$Oi = \{(TIME, (Phase1), R2)\}$$

$$Ij = \{(TIME, (Phase2), (R2, R3))\},$$

indicating that MOi uses R2 as an output only in Phase1, and MOj uses R2 and R3 as an input only in Phase2 of the cycle.

From the original tuple sets, it can be determined that MOi and MOj are data dependent, but the fact that they are also data compatible can only be recognized from the tuple sets modified to reflect times of resource usage.

No less critical an issue in I/O timing of MOs is that of taking care of multicycle operations (i.e. MOs that use machine resources for more than one MI cycle time). The construct used to deal with these operations is similar to the above "TIME" construct, and is called a "DELAY" operator. It simply indicates that use of the resource named must be delayed by the specified number of cycles. For example if the input line to the ALU, *ail*, must be allowed to stabilize for 2 MI cycles after it is set before being used, this constraint may be specified as:

$$Oi = \{(DELAY, 2, ail)\}$$

The implementation of these delays is straightforward; "dummy" MOs can be added to the SLM having the delayed resource as both input and output, to create a data dependence between the MO that uses the resource (and imposes the delay constraint) and the MOs requiring it later. For example, the sequence

MOi = <opi, Ii, ail, Ui, Ti, Fi>

MO delay = <Dummy, ail, ail, –, –, –>

MOj = <opj, ail, Oj, Uj, Tj, Fj>

ensures that the *ail*, which is used as an output by MOi, is delayed by one MI cycle time before it is used as an input by MOj. The delay MO is created to have data dependencies with both MOi and MOj, but no other conflicts occur since the U, T, and F tuple sets of the dummy MO are left empty.

### 3.3.3 Non-static Storage Resources

A **static** storage resource is one whose contents are maintained until explicitly overwritten by execution of an MO. A **non-static** storage resource, also known as a "transitory-data" or "volatile" storage resource, is one whose contents can become undefined at the end of the execution of an MI in which it has been given a value. A register is a static resource, and a microinstruction field is a non-static resource.

If one MO passes data to another via a volatile storage resource (the MOs are then considered to be **coupled**), then the MOs must reside in the same MI or data will be lost. For this reason, the concept of a **microoperation bundle** (MB) is introduced [MALL78, LAND80]; an MB is simply a set of MOs, all of which are coupled to one another.

A transformation may be performed on an SLM to make it a list of MBs by placing coupled MOs into the same MB and each uncoupled MO into its own MB. The foregoing relationships and definitions for MOs can be directly translated to be defined for MBs. For example, the definition of data dependence may be restated for bundles as:

> MBi dd MBj if there exists an MOi in MBi and an MOj in MBj such that MOi dd MOj [LAND80].

We continue, in the following chapters, to discuss compaction in terms of MOs, although the term MB may just as easily be substituted.

## 3.4 The Algorithms

The microcode compaction algorithms have recently been surveyed [LAND80], and detailed for implementation [MALL78]. Each of the algorithms proposed to date falls into one of three major categories: *linear analysis*, *critical path*, and *branch and bound*. Landskov *et al.* [LAND80] include a fourth category, *list scheduling*, but this approach is considered here to be one of the variations of branch and bound algorithms and it is dealt with as such.

Each of the categories of algorithms are described fully in the above cited references, so detailed outlines are not included here; rather only the essence of each approach is captured and the discussion deals with such issues as degree of optimality of compaction, time complexity, variations (heuristics), and some implementation considerations.

### 3.4.1 Linear Analysis

The Linear Pairwise Comparisons algorithm (LIN), first proposed by Dasgupta and Tartar [DASG76a, DASG78b], performs a sequential examination of the input SLM. This approach considers the MOs one at a time in source order and places each MO in the list of MIs by first examining data dependencies to find the earliest possible MI for placement, and then adding the MO to the first MI thereafter in which it causes no resource conflicts.

Because the LIN algorithm does not attempt to consider every legal combination of MOs in MIs for an SLM, it cannot guarantee optimality of compaction, as pointed out in Section 2.5. However, the empirical results reported in [MALL78] indicate that LIN did reasonably well (although it did not minimize) in compaction of MOs (actually MBs) into MIs, and the number of pairwise comparisons of MBs was approximately linear in the number of MBs in the SLM.

A variation on this algorithm, called LINVS for LIN with Version Shuffling, was also tested in [MALL78]. During conflict analysis for placement of an MO, each version of an MO was tested in an attempt to find a version of the MO that would fit in the MI under consideration without resource conflict. Mallett's tests show that, while it does not guarantee optimality, LINVS consistently attained minimal compaction, and this was achieved in time that was as good as, or better than, all other algorithms tested (in terms of number of pairwise comparisons of MOs).

LIN, and its variant LINVS, are reported to be the simplest of all current approaches to implement, particularly so since they do not require the construction of a DDG. Mallett concludes that LINVS is the most economical approach tested, in terms of its cost and performance.

A similar approach is used by Ma and Lewis [MA80, MA81] as part of a high-level language compiler. Their method also considers the input operations one at a time in sequential order, but it checks simultaneously for both *parallelism* and *invertibility* (data independence) of operations as the search for a position for the current MO progresses through the instructions. In addition, Ma and Lewis impose a limit on the number of MOs with which the operation to be placed may be compared. They keep track of the earliest possible instruction found to date in which the MO may be placed. When the comparison limit is reached, the MO is put in the saved instruction.

The algorithm for this approach is reasonably simple, much like the LIN algorithm in [MALL78], and Ma and Lewis show that their method packs an SLM of size n in O(n) time[6]. They also conjecture that, with a suitable choice of the maximum number of MO comparisons for any operation, code can be produced within 10 percent of the optimal.

The approach of Ma and Lewis is bound to obtain code that is further from optimal than the LIN algorithm, but if the conjecture of Ma and Lewis is true, then automatic compaction within 10 percent of optimal using LINVS should be consistently obtainable, perhaps in at most quadratic time in the worst case.

### 3.4.2 Critical Path

The critical path (CPath) approach to microcode compaction was introduced by Ramamoorthy and Tsuchiya [RAMA74]. CPath uses a DDG to create partitions of the MOs in the SLM according to the earliest and latest times when they could be executed without increasing the number of MIs. The absolute minimum number of MIs is exactly the length of longest path through the DDG, not counting edges representing weak dependencies. MOs on that path must be executed as soon as they are **data available** (i.e. as soon as the predecessor of the MO in the DDG has been placed in an MI) to minimize the number of MIs. Those MOs are identified by combining the early and late partitions to create a partition consisting of the *critical* MOs. Resource contention is examined for critical MOs and new MIs are created if necessary. Finally the non-critical MOs are placed in the MI list as early as possible in the interval between their earliest and latest timings.

As was the case for LIN, CPath does not guarantee optimality; one of its major failings in this respect is the fact that frames of the partition are split because of resource conflicts, but adjacent frames (MIs) in the list are not later considered for merging into a single MI. Mallett's findings show that CPath's performance is similar to that of LIN in somewhat longer time, and that CPath with Version Shuffling does not perform significantly better than CPath.

---

[6] They show that if m is the maximum number of MO to MO comparisons for placing any operation, then the time complexity of the algorithm is approximately n*m.

CPath is generally considered unlikely to produce near-optimal code [FISH79], and it is more complex to implement than LINVS, especially since it requires the construction of a DDG.

The local compaction techniques using "microtemplates" proposed by Tokoro *et al*. [TOKO77], is essentially the same as a CPath-type algorithm, although it considers both resource conflicts and data dependencies when forming the early and late partitions. This method effectively breaks down barriers between adjacent MIs, but the complexity of the algorithm seems to be such that it is difficult to implement. Possibly for that reason, the approach is largely ignored in the current literature, and no implementation studies were found other than those reported in [TOKO77, TOKO78].

### 3.4.3 Branch and Bound

The branch and bound (BAB) class of tree-search scheduling algorithms [AHO74, HORO78] was first applied to microcode compaction by Yau, Schowe, and Tsuchiya [YAU74].

Using a DDG, the BAB algorithm determines which MOs are currently data available based on whether all ancestors in the DDG of an MO have previously been placed in MIs. These MOs form a data available set (DAS), and the algorithm then exhaustively enumerates all possible (legal) complete microinstructions (CIs), which are MIs to which no other MO in the DAS may be added due to resource conflicts.

BAB then generates a new DAS for each CI so formed, and repeats the process for each one, thus creating a search tree of CIs. The shortest path through the final tree is an optimal list of MIs.

The search tree is generated depth-first, and the progress along any path stops when the new DAS created after construction of a CI is empty. A path may be abandoned if its length exceeds the length of the shortest path found to date, and the search may terminate if a path is found whose length equals the minimum number of MIs possible for the SLM.

This exhaustive BAB guarantees optimality, but even with the branch cutoffs described above (which do not affect optimality), the time complexity is exponential.

### 3.4.3.1 BAB Heuristics

A number of heuristics have been proposed in the recent literature to prune the search tree for the BAB approach, in many cases to the point where only one path is generated. These heuristics usually attempt to determine the "most promising" CI to expand next at any stage by assigning weights to the MOs in the DAS, and placing the most heavily-weighted MOs into an MI, avoiding resource conflicts. The CI with greatest sum of weights is chosen as the next node on the solution path.

The original Yau *et al.* paper [YAU74] proposed assigning a metric to each MO equal to the number of direct and indirect descendents of that MO in the DDG, generating all CIs, and choosing the best one to expand. This was refined in [WOOD78] to assign to each MO the number of its (direct and indirect) descendents and generating only one CI. These are both aimed at scheduling MOs with many descendents early in order to minimize data dependence bottlenecks later.

Mallett [MALL78] enumerates a variety of heuristics for ordering the choice of CI to be considered next, and for cutting off "less promising" branches of the search tree. His tests showed that the two most successful heuristics- the "most successors order" (MSO), a combination of the features of the original Yau proposal and the Wood heuristic, and "most bundles order" (MBO), which picks the CI containing the most bundles- both consistently produced optimally packed sequences of MIs, and both in time approximately linear in the number of MOs in the SLM.

Fisher [FISH79] applies a "priority list scheduling" method to the microcode compaction problem, in which MOs are ordered according to some evaluation function and then a schedule found by repeated scans of the list of prioritized MOs. His algorithm for compaction appears to be a special case of the BAB algorithm used with a heuristic; the evaluation functions used by Fisher to establish priorities among MOs may provide some good heuristics for the BAB approach.

Fisher offers some comparative empirical evaluations of his priority strategies with respect to a calculated lower bound on the number of MIs required. Unfortunately, the model of MOs upon which the strategies were tested "did not allow ... any correlation between the register and resource usages of neighboring [MOs]" [FISH79], which must be considered a severe limitation that inhibits extrapolation of these results to "real world" microcode. Fisher's "feeling" is that the resemblance of this model to the actual situation is accurate enough, and he claims that this is borne out by the fact that the relative performances of his strategies remained stable under dramatic shifts in parameter values. However, it is an inescapable fact that there *is* correlation between register and resource usages of neighboring MOs. This correlation is a critical factor, and is one of the major reasons why microcode compaction is difficult.

The methods of priority evaluation in [FISH79] are certainly worth investigation as BAB heuristics, even if the reported results may not have a basis in "real world" microcode.

### 3.5 Implementation

The algorithm implemented as part of this study was the Linear algorithm with version shuffling (LINVS). An outline of the algorithm is to be found in Appendix I, and that approach is further discussed in Chapter 6. The next two chapters describe the application to the QM-1 of the model outlined above.

## Chapter 4

## Impact of the QM-1 Structure on Compaction

The control store organization and hardware-level structure of the QM-1, or in fact of any machine, has a great deal of impact on the task of compaction.

The QM-1 control store is organized on two levels, the higher of which is made up of 18-bit words, and is referred to as the control store. The microinstructions at this level are vertical, displaying no capability for concurrency of operations. Control store operations cannot directly control hardware resources of the machine, but rather are defined by the highly horizontal words of the lower level nanostore [NANO72, SALI76]. At this lower level, a high degree of parallelism is possible between nanooperations (NOs). It is there that the local compaction technique discussed here is being applied.

Since the control store level of the QM-1 at which compaction is applied is known as "nanostore", the terminology of compaction on this machine does not correspond to that of the literature on compaction. For this reason, the terms *nanocode*, *nanooperation*, *nanoinstruction*, etc., are hereinafter used to refer to those entities called *microcode*, *microoperation*, *microinstruction*, etc., in the literature on compaction. The latter set of terms will be taken to have meaning at the higher level of control store in the QM-1, since this is the usual terminology used at that level.

### 4.1 Nanoword Structure

Nanostore consists of 360-bit wide words, each of which is divided into five 72-bit vectors (Fig. 3). The first vector, the *K-vector*, contains a number of fields and bits which serve to indicate certain conditions and operations (e.g. ALU function, shift amount, and shift direction and type), or act as scratch storage fields. The remaining four vectors, the *T-vectors*, are all identical in format. Each contains a number of fields that encode the various nanoprimitive actions of the machine. Upon activation of the T-vector, its fields are decoded and used to initiate specific hardware actions (e.g. start a main store read cycle, or gate the contents of the ALU output bus to a register).

Only one of the four T-vectors of the executing nanoword is active at any given time, while the K-vector is active throughout the execution of all of its associated

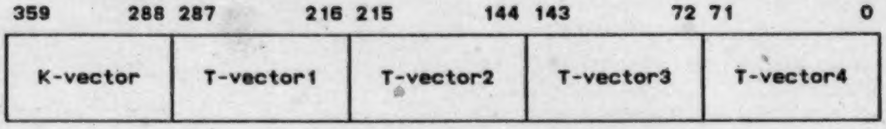


Fig. 3. Nanoword Format

T-vectors. A nanoword is read from a specific address in nanostore under program control, and must be gated into the *control matrix* from where the nanowords are executed. When this happens, the nanoword is activated and the K-vector immediately becomes active along with the first T-vector. Each of the (up to) four T-vectors is executed in sequence until another nanoword is read from nanostore and gated into the control matrix. This orderly sequence may be interrupted by a *skip* NO which suppresses the activation of the following T-step, and execution continues with the second following T-step. A new nanoword may be gated into the control matrix during any T-step and execution begins in the first T-step of that word (except if a *skip* is also commanded in the same T-step as the nanoword gate operation).

Local compaction is concerned with packing operations within a straight-line microcode segment (SLM)[7]. Since it is possible to encode a branch from any T-vector to the start of any nanoword, or a skip of any T-vector within the active nanoword (either conditionally or unconditionally), the beginning and end of an SLM can occur at any T-vector, and SLM boundaries are not restricted to nanoword boundaries. In the context of compaction, then, the combination of K-vector and the active T-vector (otherwise known as the T-step) must be viewed as the *nanoinstruction* unit. This concept, also supported in [SALI76], implies that nanoinstructions are not totally independent of one another, but rather are grouped in fours with common constant fields.

---

[7] In QM-1 terminology, the SLM is a straight-line segment of "nanocode", and is a collection of nanooperations.

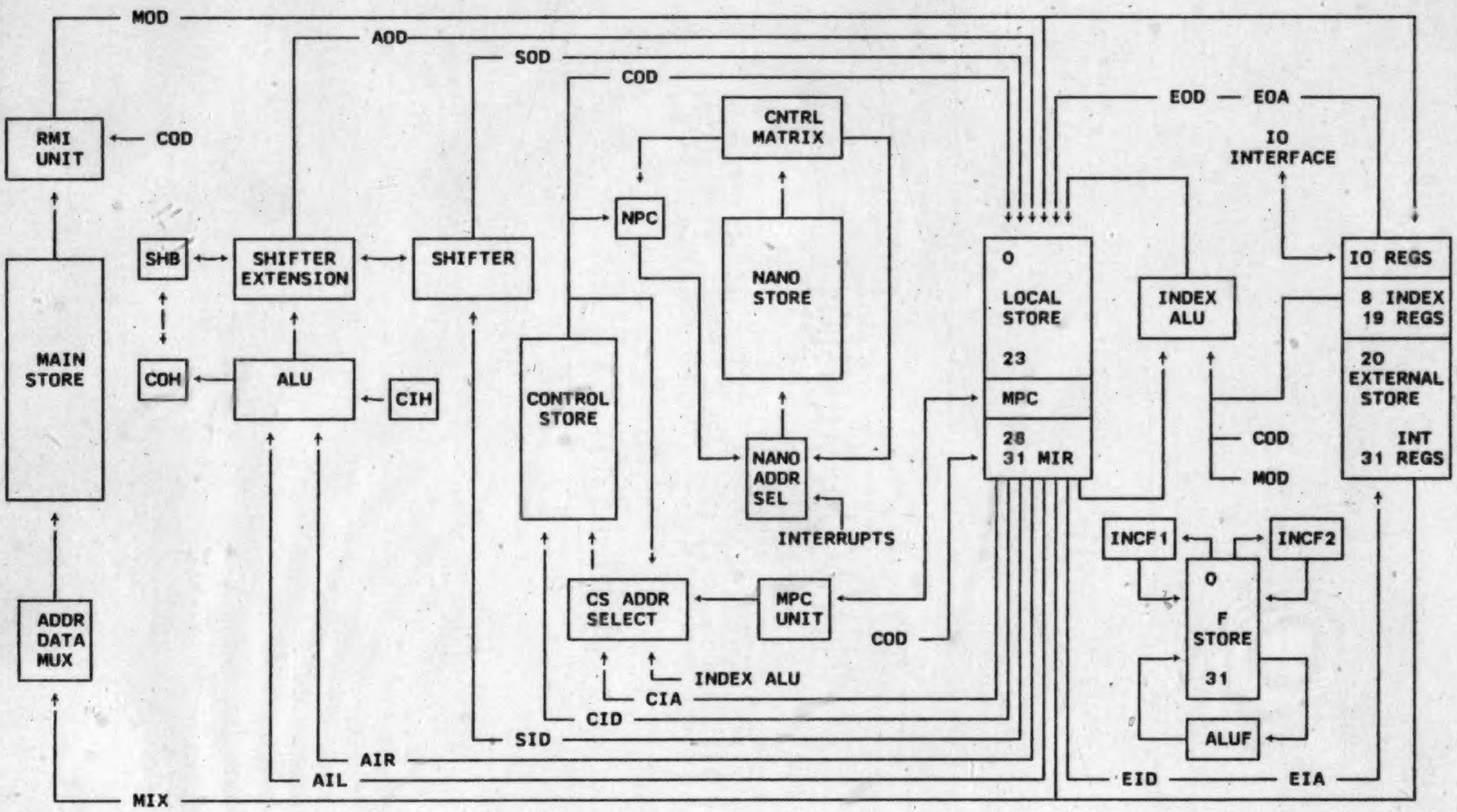

MOD
ADD
SOD
COD
EOD — EOA
IO INTERFACE
RMI UNIT
COD
CNTRL MATRIX
NPC
NANO STORE
SHB
SHIFTER EXTENSION
SHIFTER
MAIN STORE
COH
ALU
CIH
CONTROL STORE
NANO ADDR SEL
INTERRUPTS
0
LOCAL STORE
23
MPC
28
31 MIR
INDEX ALU
IO REGS
8 INDEX
19 REGS
20
EXTERNAL STORE
INT
31 REGS
COD
MOD
INCF1
INCF2
0
F STORE
31
ALUF
CS ADDR SELECT
MPC UNIT
COD
ADDR DATA MUX
INDEX ALU
CIA
CID
SID
AIR
AIL
MIX
EID
EIA

Fig. 4. QM-1 Busing Structure

### 4.2 Use of K-fields

Because each K-vector is active across up to four T-steps, efficient utilization of the K-fields is an important consideration in compaction in order that as many operations as possible may be placed in each nanoword.

Most of the 6-bit K-fields have specific uses such as function encoding or test masking, and their values are used automatically whenever certain operations are commanded in a T-step. On the other hand, all of the 6-bit K-fields may also be employed as scratch storage fields for holding values to be transferred elsewhere (e.g. to the 6-bit registers) during the execution of the nanoword.

To handle this situation in the implementation, the compactor is passed operations that specify when values in K-fields are intended for use in their assigned functions, and when it is the value (and not the source field) that is important. The compactor then has the responsibility for placing values in K-fields not otherwise used for particular functions, to maximize K-field usage and minimize the number of nanowords required for packing the NOs. This K-field assignment task is, in effect, a *resource binding* accomplished at compaction time, and it permits the implementation of the *version shuffling* feature of the LINVS compaction method [LAND80, MALL78]. Version shuffling is discussed further in Chapter 5.

### 4.3 Residual Control

The central unit of the QM-1 hardware is the *local store*, a bank of thirty-two 18-bit registers. The local store is connected, via a number of 18-bit data buses, to the other stores, register banks, and functional units of the machine (Fig. 4). Each bus may be individually connected to any local store register; this is accomplished under program control by placing the desired register number in the 6-bit F-store register corresponding to the bus. For example (Fig. 5), the left input of the ALU is selected by F-register *fail*, and the output by *faod*. The value in an F-register is static until changed by the nanoprogram and so a bus remains connected to a register as long as the F-register value is not changed. This control exercised by the F-registers over the data buses is known as "residual control" [AGRA76], which is separate from the direct control over hardware resources exercised by the encodings in the T-vectors.

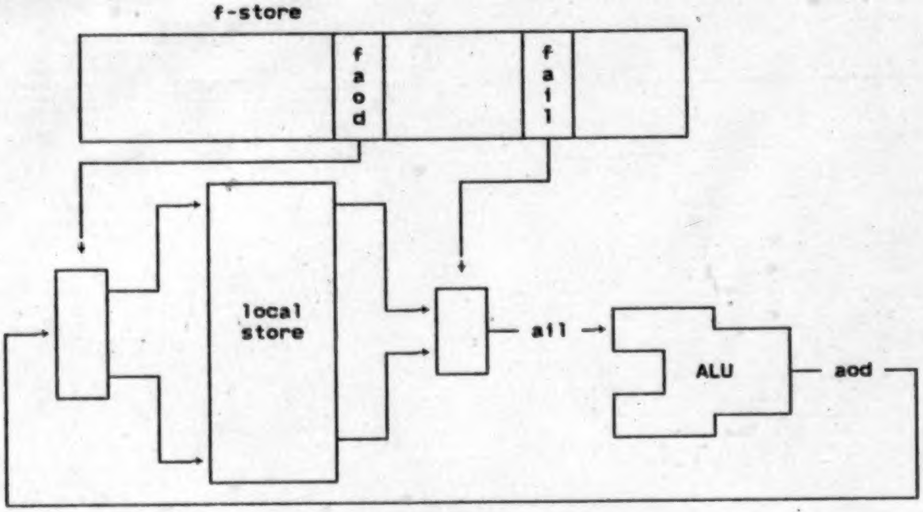


Fig. 5. Residual Control of ALU Input and Output

The F-registers are always loaded under program control, often from preset K-fields, but just as frequently from fields or resources whose contents are unknown prior to runtime. For example, any F-register may be loaded with the value of one of the arguments of the instruction in the higher level control store being interpreted.

Since all of the data buses are independent of each other and each is controlled by a separate F-register, all of the data paths may be used concurrently. It is not possible, in general, to determine whether two separate nanooperations both access the same local store register because the contents of the F-registers may be unknown. If two operations *do* access the same register (or any machine resource, for that matter), a data interaction occurs and an ordering is imposed on the operations in question. If the ordering is altered, nanocode correctness will be compromised.

Consider, for example, the following two S*(QM-1) statements [KLAS81b]. The first defines a left logical shift of the contents of local store register 3 by three bits, with the result placed in register 2; the second specifies an addition of two registers with the result placed in register 3.

**local_store[2]:=l<<3 local_store[3];**

```
local_store[3]:=local_store[5] + local_store[6];
```

Ignoring the fact that these statements actually each translate to a number of nanooperations, it is evident that the statements have a data interaction on register 3. The order of the statements then becomes important, since the result in register 2 would be different if they were reversed. (This interaction will occur, on the lower level, between the NOs that gate the shifter and ALU output buses to the respective local store registers).

A compaction-time analysis cannot always determine the exact register usage of NOs because the registers are under residual control, and so the partial ordering implied by the data interaction between NOs is also, in general, not discernible. There are two possible approaches to dealing with this problem. One is to disallow all concurrent use of local store registers (as well as external store registers, another bank of registers having multiple data paths going in and out). This effectively treats the registers as one collective resource, and is clearly an unacceptable approach since it discards much of the potential parallelism within the QM-1.

The other option is to treat all registers as independent resources, among which no data interaction can occur. This approach, adopted for QM-1 compaction, allows full exploitation of the concurrency possible in using the data buses attached to local store. However, this also introduces the possibility of allowing concurrent use of data paths in and out of local store when that is an undesirable effect. For example, an address may be calculated in the ALU and placed in a local store register for use by a subsequent main store read operation. The compactor would not be able to determine that the main store address bus is connected to the same local store register as the ALU output bus. Concurrent use of the two buses would then be allowed, resulting in a main store read operation using an incorrect address.

Since, in compacting nanocode (or microcode), the "overriding concern is preserving the algorithm programmed by the user", and the generation of a minimum sequence of instructions is the second concern [YAU74], it is necessary to introduce delays so that the above main store read operation is always executed after a gate to local store. This is true for *any* gate to local store (not just from the ALU), and the delay requirement also exists for some control store read operations. It becomes obvious that these delays are not *always* necessary and some potential concurrency is sacrificed in these cases in order to ensure correct code in *all* cases. Thus even this approach does not result in *full* exploitation of potential concurrency between operations.

Another way around the problem would be to provide a construct for specifying that one operation is to follow another. In fact, S*(QM-1) contains such a construct, called a *region*, but it is for use with low-level operations such as *gate alu*. No such construct is available for the programmer to use to specify that two higher level instructions in S*(QM-1) are to be executed sequentially, which would be taken to mean that the gating of their results is to be kept in a particular order.

### 4.4 Other Indirect Specification of Resource Usage

The problem of indirect specification of input/output resources crops up elsewhere in the QM-1 as well. For most operations, it can easily be determined which resources are to be used, since indirections are usually via T- or K-vector fields. For example, 6-bit "F-transfers" are used to load values into the residual control F-registers. The particular field or resource used as a source for such a transfer may be selected using any of the *aux* fields of the T-vector (e.g. *aux0*). The *aux0* field can also select another T-field, called *g spec*, as a source, but when this is the case, the value of the *g spec* field indicates an F-register or another field that holds the value to be transferred, rather than containing the value directly. In any case, all of the fields and/or resources acting as source or source specifiers can be determined, and so any data interactions between operations can be detected.

Unfortunately, this is not always true, such as when using the INDEX ALU, a special data manipulation unit of the QM-1 for rapid indexing and logical operations. In a "worst case" situation, choice of a local store register to supply a source value to this unit can require three levels of indirection involving two T-vector fields, a choice of a variety of F-registers or K-fields, and finally the local store register.

Figure 6 illustrates this indirection. The number of the local store register to be used as left input to the INDEX ALU resides in one of the *a* or *b* fields (subfields of local store register 31), or in *kx*, *ka*, or *kb* (which are K-vector fields). The choice of one of

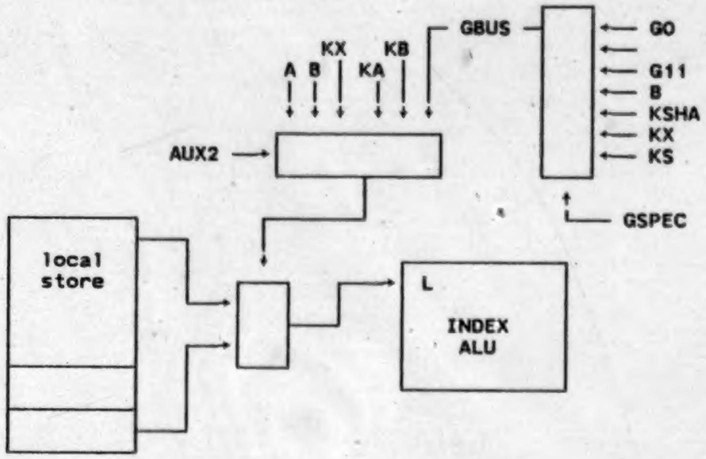


Fig. 6. Selection Hierarchy for INDEX ALU Left Input

these fields is made by the value in the T-vector field, *aux2*. Alternately, the *aux2* field may select the number on the "g bus" to indicate a local store register. The value on this bus is contained in one of the G-registers (the last twelve registers in F-store), the *b* field of register 31, or one of the K-fields indicated. The choice among these is made by the value in the T-field, *g spec*.

The first level selector fields, *aux2* and *g spec*, are T-fields and their values are always known to the compactor; the values of the fields or resources selected by them are not always known, even though the compactor is designed to trace through previous NOs for values of indirectly specified fields.

The uncertainty factor in all of this can be even more critical than simply not determining which local store register is in use. For example, the control store output bus (*cod*) can be used as an input to the INDEX ALU, but it is often impossible to tell that this is the case. The compactor is forced to *always* list the *cod* bus as an input for INDEX ALU operations; otherwise it would run the risk of placing an INDEX ALU operation using the *cod* bus as input ahead of the operation that loads the intended value onto the bus.

This kind of cautious listing of input/output resources is evident in the encodings of a number of NOs, and inevitably this will lead to placement of some NOs later in the packed SLM than actually required. This is another sacrifice made to ensure that the compactor always generates correct code. However, it is, in a sense, an expensive tradeoff. The residual control and indirections used in specifying input/output resources of NOs ultimately mean that *optimal packing of nanocode, even by exhaustive enumeration of combinations of nanooperations in nanoinstructions, cannot be guaranteed*.

The guarantee of minimal packing by methods considering all possible combinations of NOs [YAU74, LAND80], is based on the premise that each operation can be exactly specified in all respects for each instance. What constitutes "optimal" packing of NOs in a case such as the present one is an entirely different question, and is beyond the scope of this discussion. On the other hand, in order to remove or lessen the impact of such a restriction, additional constructs or other methods for specifying more exactly the resources used as input or output by each operation should be considered. Such items are discussed in the final chapter of this thesis.

## 4.5 Local Store Register 31

One local store register that may be accessed by other means than the 18-bit data buses is register 31 (R31). It contains three 6-bit fields, named *c*, *a*, and *b*, which may be used in much the same way as any other 6-bit field of the machine. In addition, R31 has a special function as the *microinstruction register* and is classified in the "control matrix domain" when serving in that capacity.

When the register is accessed in these specialized ways, it *is* possible to distinguish when R31 is in use, and extra checking is required in the compaction process to determine if the parts of it referenced are overlapping, causing data interactions. For example, an NO that reads the entire 18-bit value of R31 has a data interaction with an NO that transfers a 6-bit value to the *b* subfield. In addition, because it is possible to access R31 as an 18-bit register or as 6-bit fields (an intersection of the word-size "domains"), special processing must ensure that timing restrictions on the use of this register (or parts thereof) are not violated.

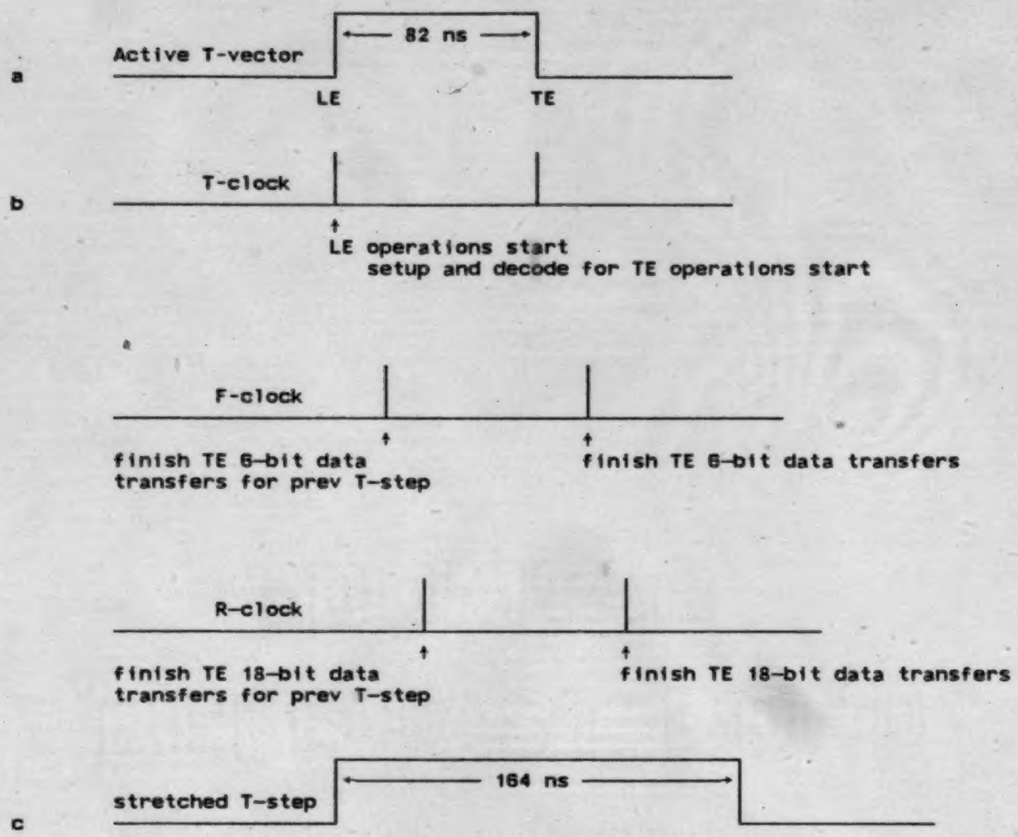

a
Active T-vector
82 ns
LE
TE
b
T-clock
LE operations start
setup and decode for TE operations start
F-clock
finish TE 6-bit data transfers for prev T-step
finish TE 6-bit data transfers
R-clock
finish TE 18-bit data transfers for prev T-step
finish TE 18-bit data transfers
c
stretched T-step
164 ns

Fig. 7. QM-1 Timing

### 4.6 Timing Considerations

Execution of NOs may be triggered at the activation of a T-step (leading edge or LE operations) or at the deactivation of the T-step (trailing edge or TE operations). The basic time between the leading and trailing edges of the main clock, known as the T-clock, is the T-period, which is 82 nanoseconds (nsec) in duration (Fig. 7a). Following the occurrence of the trailing edge of the T-clock, on which some TE operations are completed, are two other pulses, the F- and R-clocks (Fig. 7b). Other TE operations are completed on these pulses which follow the T-clock at about 20 nsec intervals and the different pulses serve to separate the 6- and 18-bit data transfers.

The time separation between NOs is measured in number of T-periods and the compactor, in accordance with the method used in the model outlined in Chapter 3, introduces "dummy" NOs between operations to ensure the proper delays in resource access. The compactor must then take into consideration the LE/TE characteristic of NOs, the particular clock upon which the NOs are executed, and another factor known as T-vector "stretching".

By setting a particular bit in a T-vector, the duration of the T-step can be *stretched* to 164 nsec (Fig. 7c) to allow additional time for propagation delays in the execution of NOs. In addition, some NOs, because of their special timing requirements, also automatically force a further one T-period extension of the T-step, called a "Phantom-stretch". Any given T-step, then, may be active for as much as 246 nsec (3 T-periods).

The timing relationships between NOs are important in packing an NO into the same T-vector as the NO upon which it is data dependent. These relationships are not simple to define because of the different manners in which NOs in the 6- and 18-bit domains interact. It is particularly difficult when the control matrix domain comes into play (e.g. when R31 is used as the microinstruction register, or when a new nanoword is gated into the control matrix).

### 4.6.1 Non-Executable Operations

Operations are defined in the input to the compactor which serve only to set K-fields or T-fields to particular values and do not actually trigger specific machine actions by themselves. For example, one operation exists to place a value in the *kalc* field which contains the ALU function specifier. The value so defined takes effect immediately upon activation of the nanoword that contains it, regardless of which T-vector contains the "operation". Such operations are defined to precede in time all other nanooperations in the same nanoword.

Special care must be taken to ensure that NOs that set K-fields are placed after NOs upon which they are data dependent; this implies that K-field set operations must be forced to a different nanoword than NOs upon which they "depend", because position in T-vectors is meaningless for these operations.

A similar situation exists for operations that only set T-fields, and so such operations are defined to precede all other nanooperations encoded in the particular T-vector, including leading edge functions.

### 4.6.2 Leading Edge Operations

Leading edge (LE) operations, which are relatively few in number in the QM-1, are defined to precede trailing edge (TE) functions in time. Where necessary, suitable delays are generated by the TE operations that use values produced by LE operations in order to ensure that enough time separates the NOs. An NO that is executed on the same clock phase and edge as another NO cannot "timing precede" that NO. It should be noted that there is no difference in clocks at the leading edge of a T-step, but the setup and decode for all TE operations begins with the activation of the T-step.

### 4.6.3 Trailing Edge Operations

In determining the timing precedence of two trailing edge NOs, their order of input is important. Given that NOj follows NOi in the input SLM, and that NOj is data dependent on NOi, NOj can only be data compatible with NOi if its clock phase is *before* that of NOi, assuming the operations are not control matrix domain NOs and thus executed on the T-clock. This definition of data compatibility is largely a consequence of the fact that NOs executed on different clock phases are in different domains; e.g. 6-bit transfers, used mainly for setting residual control registers and fields, are executed on the trailing edge F-clock, while 18-bit data transfers into and out of local store are executed on the trailing R-clock.

While data dependencies exist between NOs in the 18-bit domain and those in the 6-bit domain, it is certain (with the exceptions noted later) that the output of any NO does not cross domains.

The F-clock pulse precedes that of the R-clock, so the compactor cannot allow the TE F-clock NOs that set up the residual control for following TE R-clock NOs to be data compatible with their data dependent successors, since this would not allow enough time for the controls to stabilize before the 18-bit transfers were done. Conversely, if an 18-bit R-clock transfer is followed by a change in the residual control for the resources used in the transfer, the two operations can be defined to be data compatible. This is true, even though the F-clock pulse occurs before that of the R-clock, because, as before, the change in residual control does not have enough time to propagate far enough to affect the 18-bit value to be transferred before that transfer actually takes place.

Only three operations are executed on the trailing edge of the T-clock pulse [NANO72]. No timing problems arise with use of the *load npc* operation, and potential resource conflicts when using the *load r31* operation are detected by data dependence analysis. The third such operation, the *gate ns*, causes an undefined result in a K-field when commanded in the same T-step as an F-transfer to the K-field. Since the *gate ns* operations are entered into nanowords by the special processing routines after the actual compaction, these conflicts must be detected at that time, so no timing precedence conflicts arise between NOs executed on the T-clock trailing edge and other TE operations.

The notable exception to the domain crossing is when R31 is used in more than one of its various capacities as local store register, 6-bit fields, and microinstruction register in the same or successive T-steps. The problems encountered with respect to timing considerations surrounding use of R31 are unique and must therefore be handled separately by the compactor, but these timing relationships are well-defined [NANO72], and are therefore relatively easy to deal with.

### 4.6.4 Delay Operations

Packing of *dummy* (or *delay*) NOs is concerned with the number of T-periods between two uses of the resource in question. Delays correspond to T-periods since the execution time of an instruction may vary, and is measured in numbers of T-periods. Conceptually, the effect of the process is to insert one delay NO into each T-period between the execution of two NOs that use the resource, continuing the process until the delay NOs have been exhausted.

Any NO that uses, as an input, a resource that is delayed by a dummy NO preceding it in the SLM is data dependent upon that delay NO. In order for two such NOs to be placed in the same T-vector, they must be data compatible, and this condition depends on the duration of the T-step.

Each delay NO is defined to consume one T-period of the T-step in which it is "executed". Thus the total number of delays of any one resource placed in a T-vector must not exceed the number of T-periods consumed by that T-step. In order to add a delay or a trailing edge operation using a particular resource to a T-vector, the compactor must ensure that the T-vector already contains *fewer* delays or uses of the resources than T-periods consumed. The NO to be added "uses" the resource for one T-period.

Leading edge operations are also considered to use their output resources for one T-period. When determining the allowable number of delays of resources in a T-vector, the compactor must consider all leading and trailing edge NOs using that particular resource, as well as the number of delays already in the T-vector. Finally, NOs that only set K- or T-fields are not considered to consume a T-period.

For example, suppose NO1 is a leading edge operation that writes to resource R1; NO2 is a delay operation for the same resource, and NO3 is a trailing edge operation reading R1. If there are no other operations using R1 in T-vector T1, NO1 may be placed in that T-vector. The delay for the resource R1 may not be placed in T1 since the leading edge operation consumes one T-period. However, if the stretch bit is set in T1, NO2 may be packed into that T-vector. Finally, if NO3 is also to be placed into T1, then a further T-period must be available in the T-step, and this is only possible if T1 contains one of the operations that forces a Phantom stretch.

# Chapter 5

## Implementation of the Machine Model

Design of microcode or nanocode compactors based on a common model provides a framework for comparison of compaction implementations for various machines and algorithms, although it would be unreasonable to expect that any model would precisely fit all machines. The QM-1 nanocode compactor was designed as nearly as possible to fall within the framework of the model presented in [MALL78]. It is worth noting how this implementation compares with the model as described in Chapter 3, particularly so since one of the goals of this study is to test the applicability of that model.

### 5.1 The Tuple

The "name" of an operation exists in the QM-1 compactor implementation as an index into an internal table that encodes the NOs. This index is carried along with any instances of the NO created throughout compaction.

The I set not only lists input resources for each NO, but also any resources used as input that are indirectly specified. In the latter case, the field or resource whose value specifies the use of a resource, and the associated value or values are also listed in the I set.

To represent the conditional specification of an input resource, a new construct has been added to the model:

(COND, field__entry, field__value, resource__entry).

This is designed to specify that the resource listed in "resource__entry" is used as an input resource conditional upon the field listed in "field__entry" having the value in "field__value". In the general model, the resource__entry grouping may indicate whether the resource is static or volatile, in the same way as for other resource entries of the model (e.g. for the resource specified in a DELAY grouping). In addition, the resource__entry for a COND grouping must also be extended to allow it to contain a DELAY construct for the resource.

| name | Input set | | | | Output set | | | Timing set | | Field set | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ind | cond field | val | input resource | delay number | cond field | val | output resource | clock phase | clock edge | encoding field | val |
| 5 | | | control store | | | | cod | T | LE | rc | 1 |
| | cs addr sel | 1 | cod | | | | | | | cs addr sel | arg |
| | cs addr sel | 2 | mpc | 1 | | | | | | | |
| | | | fmpc | 1 | | | | | | | |
| | cs addr sel | 3 | mpc | 2 | | | | | | | |
| | | | fmpc | 2 | | | | | | | |

Fig. 8. NO Table Representation of 'read cs(arg)' Operation

For example, the grouping

(COND, cs addr sel, 5, (DELAY, 2, (STATIC, b)))

indicates that the *b* field of R31 (a static resource) is used as a resource, with a delay of 2 T-periods since its previous use, if *cs addr sel* (control store address select, a T-field) contains a value of 5.

Finally in the I set, for each input resource, a *delay number* is listed to indicate the number of T-periods that must elapse between the time the resource was last written or set until the time the current NO may use it as an input. The implementation of delays is discussed in Section 5.2.4.

A portion of the internal table representation for the *read cs()* operation is shown in Fig. 8. The I set includes resources listed without associated conditional fields, indicating that these are always inputs to the operation. The *read cs* NO reads from control store at an address selected indirectly by its argument. Naturally, the control store itself is an input.

Not all possible values of the 3-bit *cs addr sel* field, the argument for this NO, are listed in the example. However, when this field has value 1, the *cod* (control store output) bus is used as an input. (It acts as the source for the control store address from which to read). When the *cs addr sel* field has value 2, the microprogram counter (mpc) is an input, and since the mpc can be any one of four local store registers according to the F-register *fmpc*, the *fmpc* is also listed as an input resource. In this case, the delay number for each of these resources is 1, indicating that one T-period must elapse between the last change in either of these registers and the initiation of a *read cs(mpc)* operation.

For a value of 3 in *cs addr sel*, which corresponds to a *read cs(mpc+1)* operation, the address source is actually the output of the continuously operating mpc increment facility, which in turn requires that at least two T-periods pass between changes in the mpc or *fmpc* before the increment facility output is stable for the *read cs* to be initiated. Thus the delay numbers are 2.

It should be pointed out here, however, that since the mpc is actually a local store register and can therefore be attached to any of the stores or functional units by any residual control F-registers, it is impossible for the compactor to be certain when the mpc was last changed. Thus like most other references to the local store registers in similar situations, the mpc is actually not listed in the internal table as an input for this NO, even though it appears in Fig. 8.

The O set in the QM-1 implementation lists the output resources in a manner identical to the input resource listing in the I set, except that there is no delay information included in the O set (Fig. 8).

The use of a functional unit is implied by residual control of its inputs and outputs, and the gating of the contents of its output bus to a register. For this reason, the U set of the machine model has no meaning in the context of QM-1 nanocode compaction, and is not included in the implementation.

The timing (T) set here contains the clock phase (i.e. which of the three QM-1 clocks) and the clock edge (leading or trailing) upon which the NO is executed.

The F set (also referred to in the literature as the Residence set) contains the values of the fields in the K-vector or T-vector required to specify the operation. The description of fields in the model [LAND80] distinguishes between "control fields", which actually encode the operation (e.g. field *rc* in Fig. 8), and "immediate data fields" that supply data required by the NO, such as the location of the control store address in that same example. This distinction exists in the internal table that encodes the NOs in the form of an identifier indicating the fields that are to be filled from the parameters passed to the compactor with the index ("name") of the NO. The difference is no longer evident after an instance of an NO is created for packing.

## 5.2 Extensions

### 5.2.1 Resource Binding

The nature of the S*(QM-1) language, for which the compactor was primarily designed, requires that most resources be bound to NOs before they are passed to the compaction phase. Nevertheless, the "Orlist" construct exists in the I and O tuple sets of the implementation within the conditional specification of resources: the resources listed under the various values taken on by a particular conditional field form a disjunctive list of input or output resources.

The concept of an "Orlist" is also apparent in the creation of versions of 6-bit transfers into and out of the F-registers. For these transfers, the S*(QM-1) compiler is allowed to specify a value to be transferred rather than naming the actual source field to contain the value. The compactor is then able to choose any of a number of available source fields. For some source fields, there are as many as three sets of T-fields in which the same transfer may be encoded, leading to a substantial number of choices (and thus "versions") for each such transfer.

The "Andlist", a conjunctive list of resources, also occurs in the tabular enumeration of the resources used by each NO. Thus for the I set of Fig. 8, the tuple set may be expressed as:

$I_3$= {control store,
{(COND, cs addr sel, 1, cod) or
(COND, cs addr sel, 2,{(DELAY, 1, mpc),(DELAY, 1, fmpc)}) or
(COND, cs addr sel, 3,{(DELAY, 2, mpc),(DELAY, 2, fmpc)})
}
}

There is never a need for the "Andlist" or "Orlist" in the T set for any QM-1 nanooperation since the phase and clock edge are always fixed, and number only one phase and edge for each.

### 5.2.2 Nanooperation Versions

As pointed out in the previous section, the 6-bit F-transfers into the F-registers are very often concerned only with the transfer of a particular value and not with the actual source field that contains it. In those cases, the compactor is allowed to choose the 6-bit K-field in which to put the value required, and in which of the three sets of T-fields (F-transfer groups) to encode the operation. A version of the transfer is generated for each possibility.

The only opportunity for versions of an NO is when a K-field may be chosen as an input resource, or when there is a choice of encodings for the operation. The former situation is evident for indirect specification of INDEX ALU inputs, as well as for F-transfers; and from the point of view of the compactor, only F-transfers offer a choice of encodings.

The F-transfer operations are very frequently used in QM-1 nanoprograms. At most six transfers may be commanded in a single T-vector. Because only three distinct F-registers may be selected as sources or destinations of transfers at a time, normally only "two or three of the F-transfers are commanded in a given T-step" [NANO72]. That is, it is generally practical to have only two or three such operations per T-step.

A study of the source code of a number of QM-1 nanoprograms was carried out to determine the use of F-transfers in nanoprograms[1]. The programs included the nanocode for the QM-1 microinstruction set MULTI [NANO76], the code for a PDP11/10 emulator [DEMC76], a NOVA emulator, a cartridge tape bootstrap system for the QM-1, and an APL subscription routine. The analysis examined more than 770 nanowords, comprised of just over 2400 T-vectors. It showed that the actual average number of F-transfers per T-vector is approximately 1.5, and that in T-vectors that contained one or more F-transfers, the average number of such operations per T-vector is slightly under 2.2.

Unless K-fields are used to fullest advantage, it is certain that nanoprograms will require more nanowords than necessary. To increase the flexibility of the compactor to exploit the concurrency possible between operations, it must be free to choose, wherever possible, which K-field is to be used by an NO. Thus the binding of K-fields to operations by programmer and high-level nanoprogramming language compiler should be minimized, and such decisions delayed until compaction time wherever practical.

These factors combined make it reasonable to implement version handling capabilities for the nanocode compactor, even for these few types of operations.

### 5.2.3 Time Constraints

Viewing the nanoinstruction unit as based on the T-step, the "nanocycle" of the QM-1 is not truly multiphase. The three clocks essentially overlap and the setup and decode for functions executed at the end of the T-step begin at the same time as the initiation of operations classified as LE. For this reason, the QM-1 compactor has no need for a TIME construct attached to each input or output resource to name the phases of a multicycle clock during which the NO will require that resource while executing.

---

[1] A study on the choice of instruction encodings for the QM-1 appears in [FRIE77].

### 5.2.4 Delays

As mentioned above, each input resource for an NO has associated with it a delay number that indicates the number of T-periods to wait between uses of a resource to ensure proper timing separations between NOs. Delays are implemented as suggested in Mallett's work [MALL78] by creating dummy NOs that simply read and write the resource to be delayed; all other tuple sets are left empty for the NO.

The definition of these dummy NOs differs from Mallett's method. His model's delay operator specifies the number of instruction execution times that other operations must delay before using the resource listed. For the QM-1, the delay number refers to a number of T-periods because the duration of a nanoinstruction can vary.

A "backward" delay scheme is employed here and its use is largely a consequence of the residual control of data buses and the continuous unclocked operation of functional units of the QM-1. This organization requires, for example, that the inputs to the ALU be stable for a specified period *after* the residual control and function selection are set up before the result is stable and valid for gating or testing. A time delay is *not* required after loading values into the same F-registers before the values can be used for other purposes (e.g. transfer to K-fields, or use in the special ALU for F-registers). In the QM-1, then, a "forward" delay scheme as suggested by Mallett would result in the introduction of many unnecessary delays.

Once the dummy NOs are generated and placed in the list of NOs composing the SLM *before* the NO that caused their creation, they are packed just as any other NO, except that the timing precedence check (for establishing data compatibility) requires a somewhat different approach. The timing check is detailed in Section 4.6.4.

Whenever a delay operation is requested by an input NO, the QM-1 compactor ensures that excess delays are not generated by looking back through the list of NOs already in the SLM. When the number of delays requested exists after the last write to the delayed resource, no more delays are needed and so none are generated.

### 5.3 Volatile Resources and Bundling

Gating of a new nanoword into the control matrix causes a new K-vector to become active. Since the values of the K-fields therein may be different from those in the previous active K-vector, the K-fields may be considered to be non-static resources of the QM-1. This requires a slight variation in the definition of a volatile resource since the K-fields are not redefined after the execution of every nanoinstruction, but after execution of a group of instructions (T-steps). In the context of compaction, redefinition of a field upon activation of a new nanoword amounts to the field becoming undefined and thus the K-fields fit the definition of volatile resources in that respect.

In a way, the T-fields are also volatile resources since they become undefined (or are redefined) after each nanoinstruction. However, the only passage of data from one NO to another via T-fields is from an NO that sets one of these fields. Where possible the T-field settings are to be passed as parameters to the NOs they affect, and in other cases, the special S*(QM-1) construct, the *cocycle* [DASG78a, DASG80a, DASG80b, KLAS81b] is employed to ensure that such operations are packed in the proper T-vector. Thus there is no need for the compactor to be concerned about NOs being coupled by T-fields, and no need to bundle NOs based on T-field usage.

The grouping of nanoinstructions also forces a slight change in the concept of the *nanooperation bundle*. Rather than being a group of operations that must be placed in the same instruction, a bundle is viewed as a group of NOs that must be placed in the same nanoword.

As an example, the NOs numbered 2 and 5 in Fig. 9a must be placed in the same nanoword since NO2 puts a value into the K-field *kb*, and NO5 uses that field to transfer a value to the F-register *fsid*. If NO5 were placed in another word, the value in *kb* would be redefined (or zeroed) before the transfer.

Unfortunately, it is impossible to pack all NOs in a bundle as a single unit, as suggested in the models of the process [LAND80, MALL78], since there are often NOs between bundled NOs that are data dependent on, but not bundled with, the latter. For example, in Fig. 9a, *ka* and *kb* are scratch K-fields, and *fail* and *fsid* are residual control registers for the ALU and shifter inputs respectively. The NO sequence is bundled as in Fig. 9b, and its data dependencies are displayed graphically in Fig. 9c. Packing bundled

```
1. ka = 5
2. kb = 6
3. ka -> fail
4. gate alu
5. kb -> fsid
6. gate sh
7. ka -> fsid
```

Fig. 9a.
NO Sequence

```
Bundles =
        (1+3+7)
        (2+5)
```

Fig. 9b.
Bundles for
NO Sequence

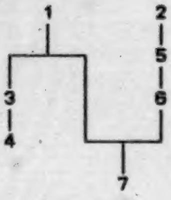


Fig. 9c.
Data Dependency
Graph for NOs

NOs as a unit implies moving them together in the input, and thus they would no longer be in source order. This can lead to incorrect code, as is evident from examining the consequences of rearranging the sequence in Fig. 9a to place all operations using *ka* together. Moving operation 7 up would result in *fsid* not being set to a new value after the gating of the shifter, and thus the next shifter operation would get its input from the wrong register. Moving operations 1 and 3 down would mean that the left input to the ALU would not be properly set before the ALU result is gated.

Bundling is accomplished by the compactor by leaving the NOs in place in the list formed from the input (containing the instances of the NOs and the generated delays), and chaining together by indices operations that must appear in the same nanoword. The chains are linked both forward and backward for easy traversal.

Associated with each nanoword during compaction is a list of NOs (by index) which must be placed in that word because their predecessors in bundles have already been packed into that word. Whenever an NO is entered in a nanoword, and it appears on this special list of NOs for that nanoword, it is replaced on the list by its bundle successor (if it has one); if it had no predecessor (and thus could not be on the list), its bundle successor (if any) is added to the list.

In this manner, the compactor ensures that all the members of a bundle are placed in the same nanoword. There can be times, however, when the position found for an NO in the nanoprogram is not in the same nanoword as its predecessor in a bundle. The NO cannot be added to that word, and further action by the compactor depends on whether the predecessor resides in an earlier or later word than that of the current NO.

If the preceding NO in the bundle is found to be in a later nanoword, an attempt is made to place the current NO into that word. Since the NO could have been placed higher in the nanoprogram for the SLM except for the bundling problem, the check for adding it into the later nanoword consists only of a residence compatibility check.

This situation can occur, for example, for the sequence of operations in Fig. 10a. The first two operations set up the residual control of the shifter input and output buses. It is assumed that these are first placed in a nanoword that also employs the K-field *ksha* for a purpose other than its special function as shift amount control (e.g. as a 6-bit scratch storage field). This same nanoword, it is assumed, does not use *kshc*, the shift control field. The *kshc* and *ksha* assignment operations of Fig. 10a would be bundled together, along with the operation that gates the shifter result to local store, since that NO requires the values in the two K-fields.

Under the assumptions outlined above, the *ksha* assignment could be placed no higher than the word following that containing the transfers to *fsid* and *fsod*, but the *kshc* assignment could be placed in the nanoword containing the F-transfers (or even earlier, depending on its previous use). This would be a violation of the requirement that the members of a bundle be placed in the same nanoword, and the predecessor (*ksha*) of the NO causing the violation (*kshc*) is to be found in a later nanoword.

In the case where the NO cannot be added to the same nanoword as its predecessor in a bundle (whether that word is before or after the word originally found for the NO to be packed), the compaction process must back up and repack the predecessor in a later word. (See Fig. 10b under the same assumptions as before.) The backing up process involves not only removing the predecessor from the nanoprogram, but also all other NOs in the same bundle that were previously packed. In addition, because NOs in the list of operations in the SLM between the members of the affected bundle may be data dependent on the bundle members, they, too, must be unpacked.

After the removal of the NOs in the backup path, the last NO removed (i.e. the removed NO that occurs earliest in the original SLM) is forced into a new nanoword at the end of the restored program, and compaction proceeds as normal from that point.

```
7->fsid
8->fsod
ksha=4
kshc=left+logical
gate sh
```

a.

```
7->fsid
8->fsod
kshc=left+logical
ksha=4
gate sh
```

b.

Fig. 10. NO Sequences Causing Bundle Exceptions

The compactor maintains a table that saves the indices of NOs causing backups in order to avoid alternately packing a sequence and backing up from the same NO indefinitely.

### 5.4 Analyses of Nanooperation Interaction

Some of the definitions of relationships between operations used in the data dependence and resource conflict analyses require revision to fit the context of QM-1 nanocode compaction.

All NOs place values in K- or T-fields of nanoinstructions by their encodings (the contents of their F sets). Many of these fields are often used as inputs or outputs by other NOs (e.g. the immediate data fields). It is therefore necessary for the compactor to consider these F set fields as output resources for the purposes of data dependence checks. Even so, the fact that two NOs require values in the same K- or T-field does not imply a data dependence between them. The NOs are then in conflict over a machine resource, and since this is detected by the residence compatibility check that the compactor performs, there is no need to compare the encoding fields of two NOs when checking for data dependence.

Because it is generally not possible to determine which local store or external store register is being accessed (the residual control problem), and since each of the register banks is therefore treated as one collective resource, input from or output to any local store or external store register is ignored in data dependence analysis.

The existence of "weak" data dependencies between NOs is determined by checking timing precedence of one NO over another. In order to establish this, the compactor must consider the clock phase and edge, as well as the stretching of T-steps.

The stretching is of particular importance when packing or checking against delay NOs, which must be processed in a slightly different way than input NOs with regard to timing considerations.

For the QM-1, *parallel* NOs may be redefined to be NOs which are data compatible and residence (field) compatible. The unit compatibility check can be ignored for resource conflict analysis since, as noted before, the U set has been left out of this implementation.

Residence compatibility, as in [MALL78,LAND80], simply determines whether two NOs require the same nanoword fields, and compares the values to be placed in common fields.

## 5.5 Summary

In summary, the QM-1 nanocode compactor was designed as much as possible within the framework of the machine model presented in [MALL78,LAND80]. Since this is a general model of machine behavior, and since machine architectures vary, there were a few departures from the model as expected.

Notable differences include the dropping of the Unit set from the tuple representation of NOs, and the consequent adjustments to the definitions of NO relationships (e.g. parallel NOs). In addition, a construct has been added for the Input and Output tuple sets for specifying resources whose use is contingent upon the value of various fields of the nanoword.

The Time tuple set includes two entries for identifying where in the cycle the NO is executed, and because there is no true multiphase cycle, the TIME construct attached to input and output resources has been eliminated.

The nature of the QM-1 set of operations and of the S*(QM-1) language leaves little opportunity for the compactor to make choices in resource bindings for NOs. The capability *does* exist for the heavily used F-transfers, and so the version shuffling extension is employed for that small subset of the QM-1 nanooperations.

Delays are handled using a "backward" delay scheme that delays the use of input resources by the current NO rather than delaying the use of output resources of the current NO by subsequent operations.

# Chapter 6

# The Linear Algorithm

The implementation of the QM-1 nanocode compactor employs the Linear Pairwise Comparisons (LIN) algorithm [DASG76a, DASG78b, MALL78, LAND80]. This algorithm performs a sequential examination of the input SLM, considering one NO at a time in source order. Each NO is placed by first examining data dependencies to find the earliest possible T-vector in which it may be placed, and adding the NO to the first T-vector thereafter in which it causes no resource conflicts [MALL78].

The choice of this algorithm was based largely on the conclusions drawn by Mallett as a result of his implementations. He states that, in terms of number of pairwise comparisons between operations, the LIN algorithm (actually the variation of LIN with Version Shuffling, called LINVS) is the "cheapest approach", and that even though it does not guarantee minimal compaction, it achieved the optimal in all of his tests. Given that LIN performs well, and that "Exhaustive comparisons of the compaction algorithms have shown that all [of the major classes] are capable of producing acceptable compactions in a reasonable amount of time" [LAND80], the decision was made to use LIN because it is, in addition, relatively simple to implement [LAND80]. This seemed appropriate for testing the feasibility of compaction of the complex QM-1 nanocode.

The flow of the general algorithm, similar to that presented in [MALL78], appears in Appendix I. The implementation for compacting nanocode consists of four major phases: reading of the input NOs one SLM at a time, including creating instances of the operations using the internal table; bundling the NOs, involving K-field use analysis and creating pointer chains; the actual compaction, similar to the general algorithm displayed in the Appendix I; and the addition of sequencing operators to the packed nanowords, with the associated bookkeeping tasks.

### 6.1 Approach of the Algorithm

The compaction phase implemented proceeds much as in the outline. Taking each NO in the order input (or generated, in the case of delay NOs), a "rise limit" is found for the NO by starting from the last nanoinstruction of the SLM and checking the NO for data dependence upon every NO in the T-vector being considered; the rise limit is the earliest T-vector in the SLM in which the NO could be placed without violating any data dependencies. The rise limit, and in fact, all positions within a nanoprogram, are expressed in terms of nanoword and T-vector.

The decision about whether an NO may be added to the T-vector at its rise limit requires a timing precedence test (to establish data compatibility) between the NO and all the NOs already in the T-vector, and a residence compatibility test to detect field conflicts. If the NO fails one of these tests, the compactor looks for the earliest possible T-vector after the rise limit to which the NO may be added; this is established solely on the basis of a residence compatibility test, since data compatibility has already been assured by means of data independence of the NO and the NOs in T-vectors below the rise limit.

The search for a position for the NO below the rise limit has an added consideration over the general approach. There are times when a new nanoword is started within an SLM before the previous nanoword has operations entered in each of its four possible T-vectors. For example, when packing two successive different ALU operations, the second will be forced to a new word regardless of where in the existing nanoword the first operation resides because of a change in *ka/c*, the K-field that specifies the ALU operation.

The compaction process may place any NO in one of these empty T-vectors below its rise limit if the NO is residence compatible with the T-vector. This implies only a test for conflict with the values in the K-fields of the nanoword. The concept of filling in "empty instructions" *within* a nanoprogram under construction does not exist in the general LIN algorithm, but by placing NOs in such T-vectors, the nanocode compactor saves space (by filling in T-vectors that would otherwise go empty, forcing more T-vectors later), and most probably saves time by packing NOs as early as possible in the SLM.

If no place for the NO is found within the existing packed SLM, it is placed either at the beginning or at the end of the SLM nanoprogram, depending on whether it has a rise limit. Placing the NO in a new first nanoinstruction involves moving the T-vectors of the first nanoword down to accommodate the NO in a separate T-vector; if the NO conflicts with the K-fields of that first nanoword, or the word already has operations in all four of its T-vectors, the nanowords must be moved down and a new one created at the beginning of the SLM. Alternately, if the start of the SLM does not coincide with the start of the nanoword (e.g. when a *skip* operation ended the previous SLM), only the T-vectors of the current SLM may be moved down to a new second nanoword when the NO is incompatible with the K-fields used by the NOs in its SLM, or the first word is full.

Before actually placing any NO into a T-vector, the check for bundle members mentioned in Section 5.6 is made. If exceptions occur, then the back up process is put into effect and compaction may be restarted from an intermediate point.

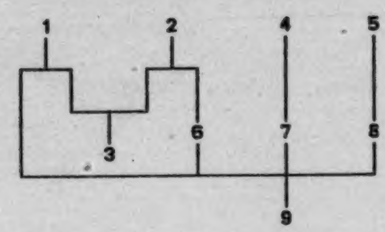
## 6.2 Version Shuffling

Versions of an NO are implemented simply as a number of different instances of the NO chained together by pointers. The handling of these versions is the same as in the general algorithm: an NO is not considered to have failed a test, such as a residence compatibility or data dependence check, until all of its versions have been found not to satisfy the particular condition being examined.

Each operation for which the compactor is allowed to choose the source K-field has encoded in its F set for each version the name and value of the source K-field. The version handler can then simply step through the versions looking for one that satisfies the condition being tested. Even though such nanooperations can use K-fields as inputs, the bundling process ignores any operations for which a choice of source is made since there is no *separate* operation to set the K-field used.

1. a->faod
2. a->fail
3. fiph->a
4. 30->fair
5. kalc=add
6. delay fail
7. delay fair
8. delay kalc
9. gate alu

a.



b.

Fig. 11. NO Sequence for an ALU operation

### 6.3 Operation of the Algorithm

The operation of the compactor is illustrated by means of the following example. The operations shown in Fig. 11a set up the residual control of the input and output buses for the *alu* (operations 1, 2, and 4), and the function selection for the *alu* (NO5). Operations 6, 7, and 8 provide the necessary time delays for the propagation of data through the *alu* before gating the result into a local store register (operation 9). The data dependencies among the operations are given in the DDG of Fig. 11b.

Each of the first three operations may be encoded in any of the three F-transfer groups, and the fourth operation can use any of a number of K-fields as a source, so each of these F-transfer operations has a number of versions.

The compactor starts by creating a nanoword and placing the first operation in the first T-vector (using the first version of that NO). The rise limit for NO2 is "non-existent", that is the operation is not data dependent on any previous NO, as can be seen in the DDG. NO2 does not conflict with NO1, since it can be encoded in another F-transfer group in the T-vector, and so it may also be placed in the first T-vector. NO3, however, is data dependent upon both of the previously packed NOs, so its rise limit is the first T-vector (T1), indicating that NO3 cannot be placed *before* T1. Because it can be encoded in the third F-transfer group, and because it does not replace the value of the *a* field before the old value is used by the previous NOs, NO3 can also go in T1. (Both NO1 and NO2 "timing precede" NO3).

```
a.     NWO:  T1.  a->faod,  a->fail,  fiph->a
             T2.  31->fair
```

```
b.     NWO:  T1.  a->faod,  a->fail,  fiph->a,  kalc=add,  delay kalc
             T2.  31->fair,  delay fail
             T3.  delay fair
```

```
c.     NWO:  T1.  a->faod,  a->fail,  fiph->a,  kalc=add,  delay kalc
             T2.  31->fair,  delay fail
             T3.  delay fair,  stretch,  gate alu
```

**Fig. 12. Stages in Packing the NO Sequence**

The fourth operation also has no rise limit, and so may be added to the first nanoinstruction found with which it has no conflicts. Since all of the F-transfer groups in T1 have been used, NO4 cannot be put there, and T1 is currently the last "instruction" in the SLM. In the general algorithm, if an operation has no rise limit, and cannot be placed in any existing instruction because of conflicts over resources or fields, a new instruction is created at the start of the SLM, and the operation added there. However, since the current nanoword has an empty T-vector (T2), NO4 is placed in T2 by the nanocode compactor[9]. The nanoprogram to this point is displayed symbolically in Fig. 12a.

NO5 has no rise limit, nor does it conflict with any of the previous NOs and so it is placed in T1. It is an operation that sets a K-field, and thus must have been bundled to any other operations that use *kalc* (i.e. operations 8 and 9).

The next three NOs all represent delays of resources to ensure the proper timing of the gating of the result to a register. NO6 has its rise limit at T1 since it depends on NO2, but since it is a delay of a resource written by a trailing edge operation (an F-transfer), it cannot be placed in T1. It has no conflicts with NOs in T2 and is placed there. NO7 depends on NO4 which was previously placed in T2, so, in the same way, it must be placed in a new instruction (T-vector) created at the end of the nanoprogram. NO8 delays a K-field set operation, and so can co-exist in the T-vector containing NO5, if no conflicts occur. This operation is put into T1, and the restrictions on the placing of the NO imposed by bundling are satisfied. Fig. 12b shows the nanoprogram at this stage.

---

[9] Possible repercussions of this implementation decision are discussed in Chapter 9.

The *gate alu* operation (NO9) has its rise limit at T3 because the delay of one of its input resources appears there. As T3 stands, NO9 cannot be added to that T-vector, because it is not timing preceded by the delay NO, which consumes the only T-period used by the T-step. However, by setting the *stretch* bit in T3, the timing compatibility condition may be satisfied (another T-period is added to the T-step execution), and NO9 may be placed in T3. The final nanoprogram for the given sequence of NOs is in Fig. 12c.

### 6.4 S* Special Constructs

The language schema S* [DASG78a, DASG80a, DASG80b] provides some special constructs that influence the packing of NOs affected by them. Two of these were found applicable for use in programming the QM-1 and were included in the S*(QM-1) language [KLAS81c].

A *region* specifies strict sequentiality of execution of the operations within the construct. NOs in a region must be placed into the nanoprogram so that they are executed in the order in which they are input, regardless of their potential concurrency. The compiler passes to the compactor control operators in the input stream to identify the beginning and end of a region (see Chapter 7), and the actual compaction of NOs within a region is quite straightforward.

To place NOs that are input in a region, the compactor simply sets an artificial rise limit for each of the operations. That limit is the later of the true rise limit found or the T-vector following the position of the previous NO in the region. An NO is thus prevented from being placed as high or higher in the SLM as its predecessor in a region.

A *cocycle* specifies that all operations within the construct must be placed in the same T-vector. Implementation of this construct in the compaction phase requires processing much as for any single NO, except that the NOs in the cocycle are all examined for each test at the same time. The NOs are packed as a unit, and no test is passed until all NOs in the cocycle are found to have a version that passes the test.

Prior to any attempt to pack the NOs in a cocycle, a check is made to ensure that all NOs within it can, in fact, be encoded in a T-vector together without conflicting over K- or T-fields. Apart from establishing that requirement, no other test (such as data dependence) is carried out among the members of a cocycle.

## Chapter 7

## The S*(QM-1) Compiler-Compactor Interface

The nanocode compactor implemented is the product of a feasibility study on compaction of QM-1 nanocode and forms a test of the local compaction theory. In addition, it is a part of the development (Fig. 1, Page 3) of a high-level microprogramming language from a language schema [DASG78a, DASG80a, DASG80b, KLAS81c]. The compiler for this language, S*(QM-1), is essentially a two-phase compiler, the second of which is the compactor.

### 7.1 Intermediate Language

The nanoprimitive command set was transformed into a representation that could be read and easily manipulated by the compactor, so that sequences of nanooperations could be input to the compactor. This was then extended, by addition of various control operators and conventions which reflect the constructs of S*(QM-1), to an intermediate language which forms the interface between the portion of the compiler that performs the translation of source S*(QM-1) code into a sequence of nanooperations in the intermediate language, and the local compactor. The extension of the compactor input to an intermediate language[10] formed part of the design process of *both* the compiler [KLAS81c] and the compactor.

The input to the compactor is a sequence of integer identifiers representing the NOs, some with optional arguments for setting specific K- or T-fields that influence the execution of the NO. These identifiers and their parameters closely parallel the nanoprimitive command set of the QM-1 [NANO72] although a few have extra information encoded as arguments for the sake of easing the task of the compactor. For example, the conditional gating of a nanostore word to the control matrix is specified in the nanoprimitive set with the value to be placed in the test specifier field. That value, in turn, determines which of three K-vector fields is used as test mask, and whether the mask is to be complemented. In the intermediate language input to the compactor, the value of the test mask field may be passed as a second parameter in addition to the test specifier.

---

[10] The intermediate language is listed and described in Appendix II.

## 7.2 Control Operators

When included in the input stream to the compactor, the various control operators give the compactor information about the NOs that follow.

S*(QM-1) allows three types of routines: instructions, subroutines, and interrupt handlers. An instruction routine specifies a nanoprogram that interprets the instructions of the higher level control store, and the first word of each such routine is designated as a legal microinstruction entry point. Subroutines may be called by instruction routines, although only a depth of one call is permitted due to the sequencing capabilities of the QM-1. Interrupt routines are programmer-defined routines to handle interrupts on the QM-1.

Each type of routine is characterized by the method of entry and exit, and by the method of sequencing through the body of the routine. A control operator exists for the start of each type and the single parameter taken by each control operator is an integer label uniquely identifying the routine for referencing.

Another control operator defines a label as an entry point of an SLM which is part of one of the three types of routines above. This operator is used to identify the location of the object of a branch.

Two more control operators denote the start of an SLM, but do not define labels for external references. One of the two indicates that normal sequencing should continue into the new SLM, and implies that the start of the new SLM may be packed into the last nanoword of the previous SLM where possible (e.g. when the previous SLM ended with a *skip* operation). The other operator forces the new SLM to start in a new nanoword,and is most useful for packing the body of an S*(QM-1) "repeat" statement.

Other control operators identify the beginning and end of the region and cocycle constructs. A final control operator serves to indicate the end of input.

### 7.3 Sequencing Considerations

Since the translation of S*(QM-1) source code produces sequential code in an intermediate language, the placing of operations for sequencing from one nanoword to another, as well as the assignment of nanostore location to the words formed, must be left for the compactor to handle.

It can be said that, since there is no requirement for the translation process to consider sequencing, the compilation from S*(QM-1) source code to intermediate code generation is relatively free from nanostore considerations, except to specify the type of routine (and hence its starting area in nanostore), and to indicate the type of sequencing to be used at the end of the SLM.

#### 7.3.1 Sequencing Within SLMs

The manner of sequencing through nanowords of an SLM may be split into two types: that for instruction routines, and that for subroutines and interrupt handlers.

Instruction routines interpret instructions of control store, and to maximize the number of possible opcodes in any instruction set in control store, only the entry word of each nanoprogram interpreting an instruction is placed on the first nanostore page. The remaining words, if any, reside on a separate page (or pages), requiring that a branch, usually an unconditional one, be set up to facilitate sequencing from the first nanoword to the second of the nanoprogram (Fig. 13). The nanoprogram counter must be loaded with the address of the object of the branch (the second word of the routine).

Sequencing through the remaining words of an SLM of an instruction is simple since the words are placed in nanostore one after another in a page separate from that containing the "header" words. Sequencing then requires only incrementing the nanoprogram counter and reading the next nanoword.

There is a reserved area for the first words of interrupt handler routines as well, and the sequencing from the first to the second words of these routines also requires a branch. However sequencing throughout both interrupt handlers and subroutines is always by means of branch operations because of the limited depth possible for calling procedures in the QM-1. These routines are called by a branch to their start addresses and return is by simply reading the next word of the calling procedure; the address of

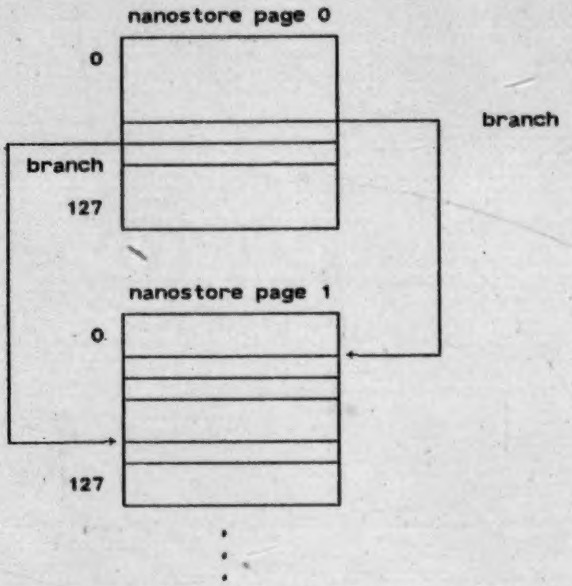


Fig. 13. Nanoprogram Organization in Nanostore

that return word is saved in the nanoprogram counter (npc), and thus called routines cannot alter the contents of the npc for sequencing.

### 7.3.2 Sequencing from the End of an SLM

The end of an SLM can occur for a variety of reasons, and the sequencing from the last word of the SLM is dependent upon that reason. The simplest passage from one SLM to the next is by falling through (e.g. from the last word of the body of an *if* statement to the first word of the SLM following the conditionally executed statements). In that case, sequencing is just as within the SLM.

In other situations, such as when calling a subroutine or executing a conditional branch, nanooperations must be inserted into the last word of the packed SLM. NOs are passed to the compactor, giving it information about the type of control transfer to be set up. These NOs are appended to the input SLM and must follow the conventions established for the interface between the S*(QM-1) compiler and the compactor. Since these sequencing NOs must appear in the last word of the packed SLM, they are stripped from the input before the regular compaction, and they are placed in the nanoprogram afterward. This approach to placing such operations was suggested by Mallett [MALL78].

The conventions specifying the control transfer NOs to be passed to the compactor, and the complementing operations that must be compactor-generated, are given in Appendix III. For example, this information for an S*(QM-1) *goto* statement is displayed in Figure 14. As is evident from the Figure, the requirements for *goto* statements in instruction routines differ from those in subroutines and interrupt handlers. In both cases, however, the *branch* bit is set in the K-vector, indicating that the next nanostore address used for reading is contained in the *kn* field of the K-vector. The nanostore address corresponding to the label passed as a parameter with the *branch* NO is placed in the *kn* field.

The compactor must generate a *read ns* NO that places the contents of nanostore at the specified address onto the nanostore bus. Since this happens to be an unconditional control transfer statement, the compactor also generates and places in the last T-vector of the SLM a *gate ns unconditional* NO that causes the word on the nanostore bus to be placed in the control matrix for execution.

The difference between *goto* statements for instructions and the other routines is in the handling of the nanoprogram counter (npc). Within an instruction, sequencing is accomplished by incrementing the npc, so that when a control transfer is commanded with no return required, the npc must be loaded with the address of the object of the branch. This allows normal sequencing to resume after the branch, and is done using the *load npc(kn)* NO in the input SLM.

On the other hand, the return address from a subroutine or interrupt handler resides in the npc and sequencing is always by means of the branch bit-*kn* field combination, termed the "nanobranch" facility [NANO72]. Thus changing the npc would destroy the return address, and so an unconditional *goto* statement within a subroutine or interrupt handler is passed to the compactor without an operation to change the npc.

Each nanoword must contain a *read ns* NO to fetch the next nanoword to be executed from nanostore and place it on the nanostore bus. This operation can be in the same stretched T-step as the *gate ns* or in any of the previous T-steps. The important consideration here is that the read operation follow any NO that changes the address for

```
Instructions:                        Subroutines and Interrupt Handlers:

NOs provided by compiler             NOs provided by compiler
    branch(label)                        branch(label)
    load npc(kn)

NOs generated by compactor           NOs generated by compactor
    read ns                              read ns
    gate ns unc                          gate ns unc


       a.                                   b.
```

Fig. 14. 'goto' Statement Sequencing Conventions

the reading from nanostore.

The *read ns* NO causes an automatic Phantom-stretch of the T-step in which it is executed. It is important, where possible, from a time-saving point of view, to place that NO in a T-vector where the extra stretching it causes does not add to the time for the nanoword. Thus an attempt is made to enter the *read ns* NO in a T-vector that already contains one of the other NOs that causes a Phantom-stretch (e.g. a *read cs*), or at least to place it in the same T-vector as the *gate ns* only if that T-vector was previously stretched. A further analysis carried out determines whether the Phantom-stretches in T-steps could be used as an alternative to setting the stretch bit for other NO timing requirements.

These considerations take into account the time for executing nanowords, and in this way the compaction process is concerned not only with reduction of nanoinstructions (or nanowords), but also with execution time reduction of nanoprograms, since the two are not exactly equivalent here.

### 7.4 S* Special Constructs

Several special constructs for denoting "micro-parallelism" were proposed in the language schema S* [DASG78a, DASG80a, DASG80b]. The *region* and *cocycle* constructs were implemented in the S*(QM-1) language [KLAS81b], but a third one, the *stcycle*, was not.

The *stcycle* specifies that the operations within it are to begin together in a new microcycle. The duration of the execution of the construct as a whole is left unspecified, since it depends on the length of execution time of the multi-cycle operations in the *stcycle*. (This is contrasted from the *cocycle* which specifies operations to begin and end in the same cycle).

Most operations of the QM-1 nano-level are executed in one "cycle" (T-step, with appropriate stretching), and those that are not (e.g. main store read) must be checked for completion by testing condition bits before the results are used. For this reason, the *stcycle* is unnecessary for programming in S*(QM-1) and so that construct was not included in the language.

# 8. The Implementation

## 8.1 The Environment

The nanocode compactor was implemented on a PDP 11/45 running the UNIX operating system [KERN81]. This environment was chosen largely because that machine already had several of the facilities related to use of the QM-1. The PDP 11/45 is linked to the QM-1 for direct loading of nanoprograms produced with the cross-assemblers that reside on the former machine. In addition, the compiler for S*(QM-1) [KLAS81c] was implemented on the same machine to take advantage of such tools as YACC and LEX that were available for compiler development [JOHN80].

The high-level programming language C [KERN78] was used to implement the nanocode compactor, since it is intimately tied to the UNIX operating system, and it provides constructs (e.g. *pointers* and *structures*) that lend themselves to the building of nanooperation lists and nanowords.

## 8.2 Strengths and Weaknesses

The nanocode compactor is able to accept sequential input code and produce correct, packed nanocode. As well as attempting to minimize the number of nanowords and T-vectors used, the program also carries out an analysis of T-step stretching before placing NOs in T-steps. This is aimed at minimizing the number of T-periods required for execution of the nanoprogram. Next to code correctness, time is the major consideration of nanocode compaction, and this analysis is an effort to gain such reductions wherever possible.

Because the compactor uses the Linear Pairwise Comparisons algorithm (LIN), its effectiveness is dependent upon the order of the input of operations, and so a sequence may attain a better packing if rearranged. An example of such a situation is given in the section containing examples of compactor performance.

The implementation is, of necessity, quite machine-dependent. The nature of the nano-level organization forces definition of "instructions" as the combination of K- and T-vectors, and hence instructions must be identified by nanoword and T-vector.

Empty T-vectors can occur within the nanoprogram being constructed and must be considered by the compactor when packing operations. The QM-1 has many "special cases" that require special processing in addition to the general algorithm. This, of course, is to be expected in any implementation.

The potential for handling versions of NOs is available in the implementation of the compactor, although not all of the possible version shuffling facilities have been included.

Perhaps the greatest weakness of this compactor, with respect to optimality of code produced, is the necessity of introducing sometimes unneeded data dependencies or delays to ensure code correctness. They are a result of a policy adopted for the compactor of including all resources when the choice of resources is unknown. The extra delays and data interactions could easily be removed if a construct were added to the S*(QM-1) language that removed the requirement for such assumptions (see Chapter 9).

### 8.3 Delay NO Considerations

Mallett points out [MALL78] that the method used to implement delay operations results in generation of dummy operations for resources that are not referenced by other operations. In his "forward" delay scheme, this can lead to the creation of instructions at the end of an SLM that contain only dummy operations.

For nanocode compaction, in which a "backward" delay scheme is used, the delay NOs for resources that must be delayed before use by another NO and are not previously set or written in that SLM, are packed at the beginning of the SLM. These delays are superfluous *within that SLM*, just as they would be at the end of the SLM, and generally cause extra T-periods to be included at the start of the packed SLM. Usually, the extra time so added to any SLM amounts to only one T-period, but, of course, the compaction process concerns itself with eliminating all extra T-periods if possible.

Even though these delays may cause extra T-periods within a given SLM, by taking a global view of the nanoprogram, it is possible to see that the delays may not be superfluous, but rather may interact with NOs in other SLMs. Recognizing when delays serve no useful purpose within an SLM would be an easy task for local compaction.

However, it would be up to a global analysis to determine whether these delays were "extra" to the nanoprogram, and, if not, to pack them where they would ensure that the necessary time separations were maintained between interacting NOs, while minimizing the duration of T-steps.

Noting (as Mallett also did) that these delay NOs may be required for delaying resource usage across SLM boundaries, the local compactor makes no attempt to remove them from an SLM.

The NO bundle is defined across the nanoword, and functional unit control is established via volatile resources (e.g. the *alu* function is defined by K-field *kalc*). Because of these factors, the same one-to-one correspondence between the time that volatile resources are defined and the period that a delay NO constrains a resource does not exist, and so delay NOs can appear in bundles. This presents no difficulty, and, in fact, the compactor packs delays in bundles exactly as it packs delays not bundled, or any other NO in a bundle.

### 8.4 Compaction Examples

Appendix IV contains some examples of the performance of the nanocode compactor. The input to the compactor in these examples is in the intermediate language given in Appendix II. As noted in the Appendix, some of the sample input was hand-translated from source S*(QM-1) code. This code is part of a microprogrammed instruction set, called QM-C [OLAF81], which is directed toward the C programming language [KERN78]. The remaining samples were hand-translated to the intermediate language from some of the nanocode that implements the MULTI microinstruction set of the QM-1 [NANO76].

### 8.5 Analysis of Compactor Performance

An analysis of the results of compaction for the QM-C instructions showed that the nanocode produced was very nearly as good as that obtainable by a nanoprogrammer. The relatively short MULTI instructions that were packed by the nanocode compactor all attained the optimum in terms of both space and time, as compared to the original nanocode of the instructions before translation to the intermediate code.

One of the reasons that the nanocode produced was not always as good as the hand-nanoprogrammed is that many SLMs were given an extra T-period duration in the first T-step. Delay NOs for resources that have no prior references in the SLM are placed at the beginning of the SLM. The result was often that the first T-step was stretched by the compactor when placing the NO that referenced the delayed resource. By looking at the code produced for the particular SLM, the reason for such stretching may not be apparent. This problem, outlined in Section 8.3, was also addressed by Mallett [MALL78], except that his extra delays accumulated at the end of the SLM.

A second way in which compaction was suboptimal is that, as pointed out throughout the literature, the performance of the LIN algorithm is dependent upon the order of input of NOs. This is evident in the example of Figures 11 and 12. If the input order were changed so that the loading of register *fair* (NO4) preceded NO3, then NO4 would have been packed into T1, the delay of *fair* (NO7) would have been placed in T2, and there would have been no need to stretch T3 in order to place the *gate alu* operation in that T-vector. (Fig. 12c). An extra T-period in the packing of that sequence of operations was directly attributable to the input order of NOs.

One method available to the nanoprogrammer for saving time and space is the dynamic loading of function selection fields for the functional units. By recognizing where he can use this ability, the nanoprogrammer can use units such as the *alu* more than once in a nanoword.

For example, in the CALL instruction for the QM-C [OLAF81] given in Appendix IV, it would have been possible, in the word at nanostore location 221, to place the value for an *alu* subtract operation in another K-field and transfer that value to *kalc* in the second or third T-step. With proper stretching of a T-step, the subtract operation of the SLM could have been completed by a *gate alu* operation in that word rather than in the following word. The subsequent NOs could then have all been packed earlier, and savings in space and T-periods would result.

The compaction algorithm is designed to pack only the operations input, and is not "intelligent" enough to recognize the possibility of using such techniques as described above. In addition, the particular algorithm used has no provision for any kind of look-ahead to later NOs in the input sequence, and this would be something of a hindrance to such analyses.

The compactor packed as well as (or nearly so, with the exceptions explained above) hand-coded nanoprograms. Examples are given in Appendix IV. The hand coding did not consider using programmed transfers to function control fields, although use of such methods can improve the code to some degree. It appears, however, that implementation of such methods in the compactor is feasible, and should be considered.

The results achieved by the compactor in the tests carried out are very encouraging, indicating that it is possible to obtain very good compaction of sequential nanocode into nanowords. This is true even over relatively long SLMs (such as the third SLM of the QM-C CALL instruction).

One of the MULTI instructions shown in Appendix IV was not packed nearly as well by the compactor as by hand-nanoprogramming. The BZS (Branch on Zero Status) instruction automatically packed takes approximately 1.5 times as long (in T-periods) as the implementation of the instruction found in the nanocode for MULTI. This is the case whether the branch is taken or not.

The major reason for this difference is that the control transfer method used in the hand-nanocoded version is not the same as the method expected by the nanocode compactor. The branch had to be rearranged when translating the MULTI nanocode to the S*(QM-1) intermediate code to conform to the conventions set up for sequencing and control transfer specification.

Since a conditional branch cannot be placed in the first word of an instruction when the alternative is sequencing to the following word, an unconditional branch to the second word (in the tail word area of nanostore) must be issued, and the conditional transfer executed there. This adds extra T-periods to the SLM. Furthermore, the local nanocode compactor does not have the global view of the code that the nanoprogrammer has, and an examination of the nanocode for the MULTI instruction shows that some operations are moved across SLM boundaries to take advantage of parallelism to provide better code.

Using the conventions for control transfers expected by the compactor (i.e. packing the intermediate code by hand), little improvement is evident for the case when no branch is taken (i.e. the condition fails), and none for when the program branches.

While the control transfer conventions seem to be largely at fault for the quality of code produced by the compactor in this instance, they were adopted to ensure correct code generation in the absence of global analysis.

# Chapter 9

# Conclusions

This study has shown that compaction of QM-1 nanocode is feasible. As the example discussed in Chapter 8 and the other tests in Appendix IV indicate, the implementation is capable of producing correct nanocode which compares reasonably well with hand-generated code.

Optimal compaction is clearly not attained, but that is a result of a combination of factors: the order-dependent algorithm used, the organization of the machine, limitations of the S*(QM-1) language, and the fact that all compactor features have not been fully implemented.

## 9.1 Applicability of the General Model

The general model of machine behavior presented in the literature [MALL78, LAND80] was found to be useful for describing the nanooperations of the QM-1. It contains constructs to adequately represent the operations and facilitate the necessary analyses for compaction. A construct was added for defining the general case of NOs that use indirectly specified resources. This lack of completeness most likely stems from the fact that residual control machines were not considered when the model was designed, but the extension of the model for these machines is quite simple.

A few of the features of the model were omitted from this implementation because they were redundant (e.g. the Unit tuple set) or otherwise unnecessary, such as the TIME constraint for resources. Others, like the delay scheme, were changed in their method of application.

The LINVS algorithm was found to be well-defined in the literature, making the basic flow easy to follow and implement.

### 9.2 Time and Space Minimization

The object of microcode compaction in general is to minimize the execution time of the microprogram, and within SLMs, this usually corresponds to minimization of the number of microinstructions. There are two major factors that separate time reduction from space reduction in the QM-1. One is the use of residual control, and the other is the make-up of the "nanoinstruction".

As Mallett observes, "a machine with residual control may be programmed to have a loop achieving a time reduction at the expense of employing more microinstructions" [MALL78]. This can be seen to be the case for the QM-1 by observing Fig. 15. Figure 15a gives the function of a loop in S*(QM-1) code. Its task is to add a series of local store registers into register 30. The series begins at the register whose number is in the residual control register *fair*, and additions continue until the value in *fair* is decremented to zero.

The implementation of the loop in one nanoword (Fig. 15b), which includes the *alu* control setups, requires 5 T-periods per iteration. If the residual and functional control setup operations are taken out of the loop, sacrificing an extra nanoword, each iteration of the loop requires only 4 T-periods. Since the setup word requires four T-periods, time is saved after four executions of the loop by using this extra nanoword.

Considering the fact that once any operation is entered into a T-vector of a given nanoword, the entire 360 bits of the word are "in use", the nanocode compactor can save space only by reducing the number of nanowords used. No storage saving is obtained by reduction of the number of T-vectors used in any nanoword, but a time saving is possible by ensuring that T-steps are not stretched more than necessary. No time is lost if all four T-vectors are not used in a nanoword, but the compactor makes efficient use of the available T-vectors leading to a reduction of the total number of nanowords required, and thus a space reduction.

In terms of the actual results of the compaction tests, it is apparent that the implemented compactor is capable of consistently producing nanocode that is within twenty-five percent of hand-generated code. Since full version shuffling capabilities were not implemented, and various tradeoffs were made to ensure correct code while supporting high-level microprogramming in S*(QM-1), these results are in line with

```
repeat
  local_store[30]:= local_store[30] + local_store[fair];    /* add regs */
  fair:= decl fair;                                         /* decr fair */
until (fair==0)
```

Fig. 15a. Loop in S*(QM-1) Code

```
word n:
 .... ka=30, kx=f zero, kalc=add                  /* constants required */
      branch(n.+1)                                /* addr of next word */

 X... ka->fail, ka->faod                          /* set up alu resid ctl */

 .S.. gate alu                                    /* result to R30 */
      decf->fair                                  /* decr fair */
      f(fair), skip(not x)                        /* no read if fair not 0 */

 ..X. read ns                                     /* read word after loop */

 ...X gate ns                                     /* gate word to execute */


      /* loop in one NW requires 5 T-periods */
```

Fig. 15b. Loop with Residual Control Inside Loop Nanoword

Mallett's reported findings [MALL78, MA81].

### 9.3 Machine Organization

Some of the organizational features of the QM-1 have a large impact on the compaction process. The nanoword format, as noted above, causes a difference in the analyses required to reduce time and space use for an SLM.

Because of the frequent use of the K-fields in their various capacities, it might appear that more of them would be useful, particularly from the point of view of the compactor. However, it appears that the compactor rarely runs out of scratch storage K-fields for placing values to be transferred to F-registers, even when some fields are

```
word n:
 .... ka=30, kalc=add                              /* constants required */

 X... ka->fail, ka->faod                           /* set up alu resid ctl */
      load npc(seq)                                /* incr npc */

 .S.. read ns, gate ns                             /* read & gate loop word */


word n+1:
 .... kx=f zero, kalc=add                          /* constants required */
      branch(n.+1)                                 /* addr of next word */

 S... gate alu                                     /* result to R30 */
      decf->fair                                   /* decr fair */
      f(fair), skip(not x)                         /* no read if fair not 0 */

 .X.. read ns                                      /* read word after loop */

 ..X. gate ns                                      /* gate word to execute */


  /* loop in two NWs requires 4 T-periods for setup, 4 for iterations */
```

Fig. 15c. Loop with Residual Control Outside Loop Nanoword

used as test masks or for functional unit control. More often, new nanowords are forced by operations specifically requiring the same fields as prior operations. For instance, successive *alu* operations performing different arithmetic or logical functions require different values in the *kalc* field.

The alert nanoprogrammer is sometimes able to save nanowords by such means as loading a new value into *kalc* from another K-field, thus permitting two different *alu* operations in one nanoword. The S*(QM-1) compiler cannot see the placing of operations into words, and the compactor is designed to pack the operations that it is given, so it is not possible in this context to utilize such programming techniques. The generation of such operations moves into the realm of global analysis and automatic microprogram generation, and is beyond the scope of this study.

### 9.3.1 Using Empty Instructions

As noted in Chapter 6, the nanocode compactor must deal with T-vectors left empty at the end of nanowords. It was decided that the compactor should place nanooperations into such T-vectors whenever one was encountered during the search for a position for an operation. In general, this approach results in the operation being placed in its highest possible position in the current SLM, since resource conflicts prevent its placement higher in the SLM, and the search for position continues downward.

However, this is not necessarily the case if the operation to be placed is data dependent on no other NO in the SLM, and conflicts over resources with an NO in every "used" T-vector in the SLM. The LIN algorithm then normally places such an NO in its own separate instruction at the beginning of the SLM, in order to allow other operations that are dependent upon it to be placed as early as possible in the SLM. The nanocode compactor, rather than doing this, places the NO in the first empty T-vector in an existing nanoword, and only puts it at the beginning of the SLM if there are no empty T-vectors in the SLM.

In the worst case, an NO could be placed in a T-vector of the last nanoword of the SLM (at the end of the current nanoprogram), if there were no preceding empty T-vectors. This eventuality is unlikely to occur often since, based on the packing tests, empty T-vectors are generally frequent (nanowords often use the last of the T-vectors only for sequencing operations, such as *gate ns*), and NOs rarely conflict with operations in all T-vectors of nanoprograms of any length. The possibilities for generating less efficient code by filling in these empty T-vectors when possible is more than offset by the potential gains in savings of nanoword space and of time.

### 9.3.2 Higher Level Concurrency Specifiers

The effect of the residual control feature forcing extra delays into an SLM, where they might not be needed, has been discussed. One of the solutions to this problem, as pointed out in Chapter 4, would be to provide special constructs to the S*(QM-1) programmer to allow him to pass to the compiler knowledge that he possesses about the interactions of the S*(QM-1) statements in his program.

```
f_store.fcia=4;
seqbegin
  local_store[4]:=local_store[3]+instruction_reg;
  control_store.output:=control_store[f_store.fcia]
seqend;
```

Fig. 16. Sequence Construct

```
parbegin
  local_store[5]:=1<<3 local_store[3]  □
  local_store[5]:=local_store[4]+local_store[6]
parend;
```

Fig. 17. Parallel Construct

For example, the compactor is forced to assume that an operation reading from one of the stores, using an address stored in a local store register (e.g. *read cs(cia)*), is dependent upon a preceding gate operation to local store (e.g. *gate alu*) to place the address in the register. This is often the case, but the compaction process has no way of determining for certain whether or not it is. On the other hand, the programmer does know which instructions are used to calculate addresses for subsequent control store or main store read operations, and so a method of passing that information to the compiler, without forcing him to program at the nanooperation level, is required.

An analagous situation exists for specifying operations to be performed concurrently, although the occurrence of this type of situation is less frequent than the need to specify sequentiality of instructions[11]. A higher level construct than the *cocycle* would allow the specification of such concurrencies without low-level nanoprogramming.

Two "high-level" constructs are proposed here for inclusion in the S* language schema, and in particular in the language S*(QM-1).

---

[11] The programmer may, for example, want to take advantage of the fact that simultaneous transfers to the same local store register results in the logical 'OR' of the values transferred.

To specify sequentiality of instructions, the *sequence* construct is introduced, to be specified in S*(QM-1) programs with delimiters *seqbegin* and *seqend*. This construct tells the compiler that the instructions within must be *completed* in the order they were input (Fig. 16). This implies that the residual control setups may be done in any order, but that the operations that actually place the results of the instructions in the output resources must be kept in sequence as input. The compiler is then required to translate the instructions to nanooperations, placing the NOs that complete the instructions into a *region* to ensure that they are executed sequentially. The other operations may be packed to take advantage of the possible concurrencies among operations. For Fig. 16, a *gate alu* NO and a *read cs* NO represent the "completion" operations and so must be placed in a *region*.

Similarly, the *parallel* construct, specified by means of the delimiters *parbegin* and *parend*, forces the "critical" operations into a *cocycle*. The *gate sh* and *gate alu* NOs produced from the instructions of Fig. 17 would be placed in a *cocycle*. Any nanooperations not placed inside these constructs would be assumed to have no special order requirements, and would be packed according to the compaction algorithm considering the usual criteria.

These constructs, in allowing the programmer to identify interactions among high-level instructions, not only free him from considering low-level operations, but also permit the compactor to rely on the compiler output to reflect the necessary information about NO relationships. The artificial delays and data dependencies introduced to ensure code correctness may be removed since the programmer is given the ability (and responsibility) to properly identify the interacting operations. In this way, the amount of information lost in translation of the algorithm the programmer had in mind to intermediate code is reduced, and the quality of compaction can be improved.

The *sequence* construct can also be used to remove inefficiencies caused by indirect specifications of resources. Using it, the programmer can inform the compactor when the control store output bus is used as in input to the INDEX ALU. The compactor will no longer be required to introduce artificial data dependencies by always listing the bus as an input to such operations, and better packing will result.

```
new instruction (label 0)
mpc plus 1
31->fail
29->faod
kalc=pass left
gate alu
26->fmpc
28->fcia
26->fcid
read cs(mpc+ab)
31->fcod
gate cs
eof
```

a.

```
word0:
 .... legal micro op, branch(220),
      ka=31, kb=29, kx=28, kt=26,
      kalc=pass left
 S... load npc(kn), kt->fmpc, mpc plus 1,
      ka->fail, kb->faod
 .P.. read ns, gate alu,
      kx->fcia, kt->fcid, ka->fcod
 ..X.
 ...S read cs(mpc+ab), gate cs, gate ns

   1 nanoword, 8 T-periods
```

b.

Fig. 18. SLM Without Cocycle

```
new instruction (label 0)
mpc plus 1
31->fail
29->faod
kalc=pass left
gate alu
cocycle
  26->fmpc
  28->fcia
  26->fcid
coend
read cs(mpc+ab)
31->fcod
gate cs
eof
```

a.

```
word0:
 .... legal micro op, branch(220),
      ka=31, kb=29, kalc=pass left
 S... load npc(kn), mpc plus 1,
      ka->fail, kb->faod
 .S.. read ns, gate alu, gate ns

word220:
   .... ka=26, kb=28, kx=31
   X... load npc(seq), ka->fmpc,
        ka->fcid, kb->fcia
   .P.. kx->fcod, read ns
   ..S. read cs(mpc+ab), gate cs,
        gate ns

   2 nanowords, 11 T-periods
```

b.

Fig. 19. SLM With Cocycle

## 9.4 Using S* Constructs

The compactor must be given as much information as possible about interactions between operations, and this may be accomplished, in part, by use of the constructs provided by the S* schema and the S*(QM-1) language. However, use of these constructs where not required hampers the effectiveness of the compactor by forcing it to place operations where they might not result in the best packing possible.

For instance, the sequence of operations in Fig. 18a is the first SLM of the *call* instruction of the QM-C instruction set [OLAF81]. Figure 18b demonstrates that this sequence of operations can be packed into one nanoword, requiring 8 T-periods to execute. In Fig. 19a, three F-transfers are arbitrarily placed in a *cocycle* to force them into the same T-step. This is a "legal" use of a *cocycle*, but inclusion of it where unnecessary results in a packing requiring two nanowords with total duration of 11 T-periods (Fig. 19b), a severe degradation.

Without constructs such as the *cocycle*, code correctness cannot be guaranteed, but misuse of the constructs will lead to less efficient code. The programmer must be careful to use them only where absolutely necessary.

### 9.5 Size of SLMs

Wood, in his paper on packing of microoperations [WOOD78], states that SLMs "are typically not very long", and claims that "sequences of more than about ten statements without a jump are uncommon, and the data inter-dependency in such cases is liable to be great."

Observations made during the course of the present study do not support Wood's contention about size of SLMs. In particular, in hand-translating some of the instructions of the QM-C instruction set [OLAF81] from S*(QM-1) code to the intermediate representation, it was found that SLMs actually tend to be a minimum of Wood's ten operations, and can be much longer. This large difference in size is probably attributable to the use of residual control in the QM-1, which requires a significant number of operations to set up.

Use of residual control means that there does exist a fair amount of "data inter-dependency" among nanooperations, but neighboring operations in an SLM always exhibit a certain amount of correlation in their resource usage. Residual control increases the potential parallelism of operations in the machine and thus does not hamper the compaction, even if it does increase the amount of dependence among operations somewhat.

The length of SLMs of nanooperations in the QM-1 will serve to increase the impact of local compaction applied above, although it is probable that global analysis will also be important, especially for efficient use of residual control over resources of the QM-1.

## 9.6 Time Complexity of the Implemented Compactor

This study presents no figures on the time complexity of the compaction process, which is usually expressed in terms of the number of pairwise comparisons between operations.

One of the difficulties for gathering data to determine the time complexity is that it is not exactly clear what constitutes a pairwise comparison of nanooperations. For example, part of the resource conflict analysis of compaction is the checking for contention over K-vector fields in a nanoword. This analysis is carried out by checking a list of values already in the K-fields, rather than by comparison to individual NOs.

It becomes necessary, then, to define pairwise comparisons between operations within the framework of the QM-1 organization, or at least of the organization of the representation of operations and instructions. Such a machine-dependent definition of comparisons makes time complexity data useless, except possibly for comparisons to other algorithms implemented for QM-1 nanocode compaction.

The concern of this study was not to test the efficiency of the algorithm used, but rather to apply the general machine model, and to investigate the feasibility of nanocode compaction. Reasonably efficient code produced in less than exponential time is acceptable, and was obtained.

## 9.7 Topics for Further Study

The nanocode compactor implemented as part of this study is able to properly compact sequences of nanooperations into nanowords. However, some work remains to be done to complete the compactor itself. In particular, the version shuffling facility has only been partially implemented, and completion of this portion of the program would help in obtaining more efficient code. The implementation requires a certain amount of fine tuning to ensure that it is saving as much time and space as possible. The compactor is not able to handle the S*(QM-1) *case* statement. This will involve some nanostore allocation and some sequencing considerations.

One as yet unresolved problem in compaction of nanocode is related to residual control and the use of local store register 31 (R31). If, for example, R31 is used as an input to the ALU (i.e. the value 31 is placed in *fail* or *fair*), and another operation clears one of the subfields of R31, then a data dependence occurs between each of these operations and the *gate alu* operation. As with other such NOs that depend upon residual control, it may not be possible to determine when R31 is to be used as an ALU input, and so these critical dependencies may go undetected.

In addition to, or as part of, the S*(QM-1) translation phase of the compiler, an optimization phase is desirable to produce reduced intermediate code by elimination of redundant and nonessential operations. This would ensure that the compactor would only pack the necessary operations, and should result in significant time and space savings.

A global compaction phase to analyse the code produced by the local compaction would be beneficial. This process would require that local analysis retain and pass on more information than it currently does (e.g. identification of start of subroutines), but that would present no difficulty. The global analysis would face such problems as timing considerations across T-steps whose duration can vary[12], and the rearrangement of sequencing operations when eliminating T-vectors or nanowords. It should also consider generating NOs for dynamic loading of function specification fields, such as *kalc*.

As pointed out in Chapter 8, delay NOs for resources for which there is no prior reference within the SLM generally cause an extra T-period to be added at the start of the SLM. Global compaction would distribute these delays where needed just as it would for a "forward" delay scheme where such delay NOs accumulate at the end of the SLM.

Finally, the nanocode compactor must be extensively tested and its results compared to hand-generated nanocode, in order to assess its effectiveness. Independent of that evaluation, the compactor should be analysed as part of the entire process of generation of nanocode from a high-level nanoprogramming language.

---

[12] A *skipped* T-step consumes one T-period before the start of the subsequent T-step; the same T-step, if not *skipped*, may be up to 3 T-periods in duration.

## Bibliography

AGER76 Agerwala,T., "Microprogram Optimization: A Survey", *IEEE Trans. Comput.* **C-25**, 10 (Oct. 1976), p.962

AGRA76 Agrawala,A.K., Rauscher,T.G., *Foundations of Microprogramming*, Academic Press Inc., New York, 1976

AHO74 Aho,A.V., Hopcroft,J.E., Ullman,J.D., *Design and Analysis of Computer Algorithms*, Addison-Wesley Pub. Co., Reading, Mass., 1974

DASG74 Dasgupta,S., "A High-Level Microprogramming Language", M.Sc. Thesis, Dept. of Computing Science, University of Alberta, 1974

DASG76a Dasgupta,S., Tartar,J., "The Identification of Maximal Parallelism in Straight-Line Microprograms", *IEEE Trans. Comput.* **C-25**, 10 (Oct. 1976), pp.986-992

DASG76b Dasgupta,S., "Parallelism in Microprogramming Systems", Ph.D. Dissertation, Dept. of Computing Science, University of Alberta, 1976

DASG77 Dasgupta,S., "Parallelism in Loop-Free Microprograms", *Information Processing 77 (Proc. IFIP Congress)*, **7**, B.Gilchrist, Ed., North Holland Pub., Amsterdam, 1977, pp. 745-750

DASG78a Dasgupta,S., "Towards a Microprogramming Language Schema", *Proc. 11th Annual Workshop on Microprogramming (ACM)*, Nov. 1978, pp.144-153

DASG78b Dasgupta,S., "Comments on the Identification of Maximal Parallelism in Straight-Line Microprograms", *IEEE Trans. Comput.* **C-27**, 3 (March 1978), pp.285-286

DASG79 Dasgupta,S., "The Organization of Microprogram Stores", *ACM Comput. Surv.* **11**, 1 (Mar 1979), pp.39-65

DASG80a Dasgupta,S., "Some Implications of Program Methodology for Microprogramming Language Design", *Proc. IFIP TC-10 Conference on Microprogramming, Firmware, and Restructurable Hardware*, G.Chroust and J.Mulbacher, Eds., North-Holland Publishers, Amsterdam, 1980, pp.243-252

DASG80b Dasgupta,S., "Some Aspects of High-Level Microprogramming", *ACM Comput. Surv.* **12**, 3 (Sept 1980), pp.295-323

DEMC76 Demco,J.C., Marsland,T.A., "An Insight into PDP-11 Emulation", *Proc. 9th Annual Workshop on Microprogramming(ACM)*, Sept 1976, pp.20-26

DEWI76 DeWitt,D.J., "A Machine Independent Approach to the Production of Horizontal Microcode", Ph.D Dissertation, University of Michigan, Ann Arbor, June 1976; *Tech. Rpt. 76 DT4, Aug. 1976*

DOMA75 Domaschenko,J., "An Implementation Study on Microprogram Optimization", M.Sc. Dissertation, Dept. of Computing Science, University of Alberta, 1975

ECKH71 Eckhouse,R.H., "A High Level Microprogramming Language (MPL)", *Proc. AFIPS SJCC* **36**, AFIPS Press, Montvale,N.J. (1971), pp.169-177

FISH79 Fisher,J.A., "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources", Ph.D. Dissertation, Dept. of Mathematics and Computing, NYU, Oct. 1979

FRIE77 Frieder,G., Miller,J., "An Analysis of Code Density for the Two Level Programmable Control of the Nanodata QM-1", *Proc. 10th Annual Workshop on Microprogramming(ACM)*, Sept 1977, pp.22-32

HORO78 Horowitz,E., Sahni,S., *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Maryland, 1978

JOHN80 Johnson,S.C., "Language Development Tools on the Unix System", *Computer* 13 8, 1980, pp.16-22

KERN78 Kernighan,B.W, Ritchie,D.M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978

KERN81 Kernighan,B.W, Mashey,J.R., "The UNIX Programming Environment", *Computer* 14, 4, 1981, pp.12-24

KLAS81a Klassen,A.B., Dasgupta,S., "S*(QM-1): An Instantiation of the High Level Microprogramming Language Schema S* for the Nanodata QM-1", *Tech. Rpt. TR81-4*, Dept. of Computing Science, University of Alberta, June 1981

KLAS81b Klassen,A.B., Dasgupta,S., "Syntax and Semantics of the High Level Microprogramming Language S*(QM-1)", *Tech. Rpt. TR81-3*, Dept. of Computing Science, University of Alberta, August 1981

KLAS81c Klassen,A.B., "S*(QM-1): An Experimental Evaluation of the High Level Microprogramming Language Schema S* using the Nanodata QM-1", M.Sc. Thesis, Dept. of Computing Science, University of Alberta, 1981

LAND80 Landskov,D., Davidson,S., Shriver,B., Mallett,P.W., "Local Microcode Compaction Techniques", *ACM Comput. Surv.* 12, 3 (Sept. 1980), pp.261-294

LEE74 Lee,J.A.N., *The Anatomy of a Compiler*, D. Van Nostrand Company, New York, 1974

KLEI74 Kleir,R.L., "A Representation for the Analysis of Microprogram Operation", *Proc. 7th Annual Workshop on Microprogramming (ACM)*, Sept. 1974, pp. 107-118

MA80 Ma,P.-Y., Lewis,T.G., "Design of a Machine-Independent Optimizing System for Emulator Development", *ACM TOPLAS* **2**, 2, April 1980, pp.239-262

MA81 Ma,P.-Y., Lewis,T.G., "On the Design of a Microcode Compiler for a Machine-Independent High-Level Language", *IEEE Trans. Software Engineering"* **SE-7**, 3, May 1981, pp.261-274

MALL78 Mallet,P.W., "Methods of Compacting Microprograms", Ph.D. Dissertation, University of Southwestern Louisiana, Dec. 1979

NANO72 *QM-1 Hardware-Level Users Manual*, Nanodata Computer Corp., Buffalo, N.Y., 1972; Third Edition, Revision 1, Sept. 1979

NANO76 *MULTI Micromachine Description*, Nanodata Computer Corp., Buffalo, N.Y., Revision 1, March 1976

OLAF81 Olafsson,M., "The QM-C: A Microprogrammed C-Oriented Instruction Set for the Nanodata QM-1", M.Sc. Thesis, Dept. of Computing Science, University of Alberta, 1981

PATT76 Patterson,D.A., "STRUM: Structured Programming System for Correct Firmware", *IEEE Trans. Comput.* **C-25** , 10 (Oct 1976), pp.974-985

POE80 Poe,M.D., "Heuristics for the Global Optimization of Microprograms", *Proc. 13th Annual Workshop on Microprogramming (ACM)*, Nov. 1980, pp.13-22

RAMA74 Ramamoorthy,C.V., Tsuchiya,M., "A High Level Language for Horizontal

Microprogramming", *IEEE Trans. Comput.* **C-23**, 8 (Aug. 1974), pp.791-801

SALI76 Salisbury,A.B., *Microprogrammable Computer Architectures*, American Elsevier Pub. Co. Inc., New York, N.Y., 1976

SALO76 Salomon,D., "A Sequential Microprogramming Language", M.Sc. Dissertation, Dept. of Computing Science, University of Alberta, 1976

SITT73 Sitton,W.G., "Strategies for Microprogram Optimization", Ph.D. Dissertation, Dept. of Computing Science, University of Alberta, 1973; *Tech. Rpt. TR73-4*, April 1973

SRIM80 Srimani,P.K., Sinha,B.P., "Some Studies on Microprogram Optimization", *Proc. 13th Annual Workshop on Microprogramming (ACM)*, Nov. 1980, pp.30-37

TOKO77 Tokoro,M., Tamura,E., Takase,K., Tamaru,K., "An Approach to Microprogram Optimization Considering Resource Occupancy and Instruction Formats", *Proc. 10th Annual Workshop on Microprogramming (ACM)*, Oct. 1977, pp.92-108

TOKO78 Tokoro,M., Takizuka,T., Tamura,E., Yamaura,I., "A Technique of Global Optimization of Microprograms", *Proc. 11th Annual Workshop on Microprogramming (ACM)*, Nov. 1978, pp.41-50

WOOD78 Wood,G., "On the Packing of Micro-Operations into Micro-Instruction Words", *Proc. 11th Annual Workshop on Microprogramming (ACM)*, Nov. 1978, pp.51-55

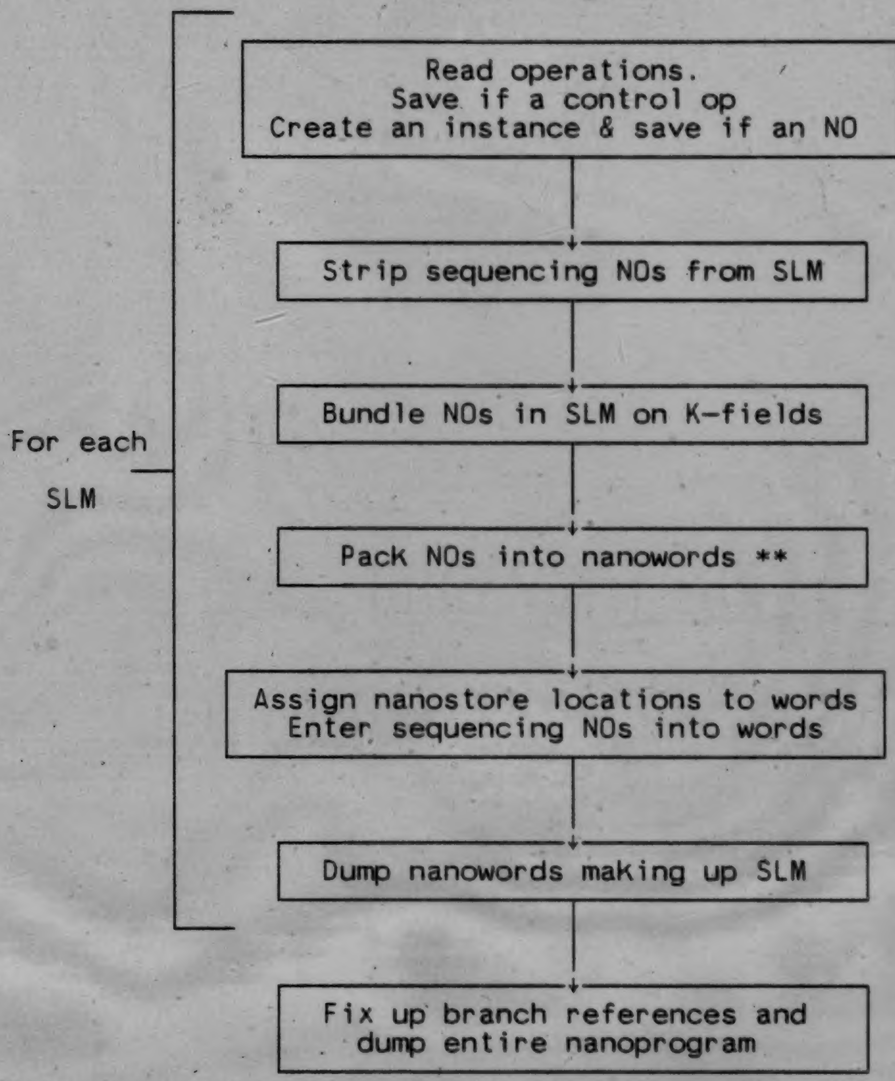
WOOD79 Wood,G., "Global Optimization of Microprograms through Modular Control Constructs", *Proc. 12th Annual Workshop on Microprogramming (ACM)*, Nov. 1979, p. 1

YAU74 Yau,S.S., Schowe,A.C., Tsuchiya,M., "On Storage Optimization of Horizontal Microprograms", *Proc. 7th Annual Workshop on Microprogramming (ACM)*, Sept. 1974, pp.98-106

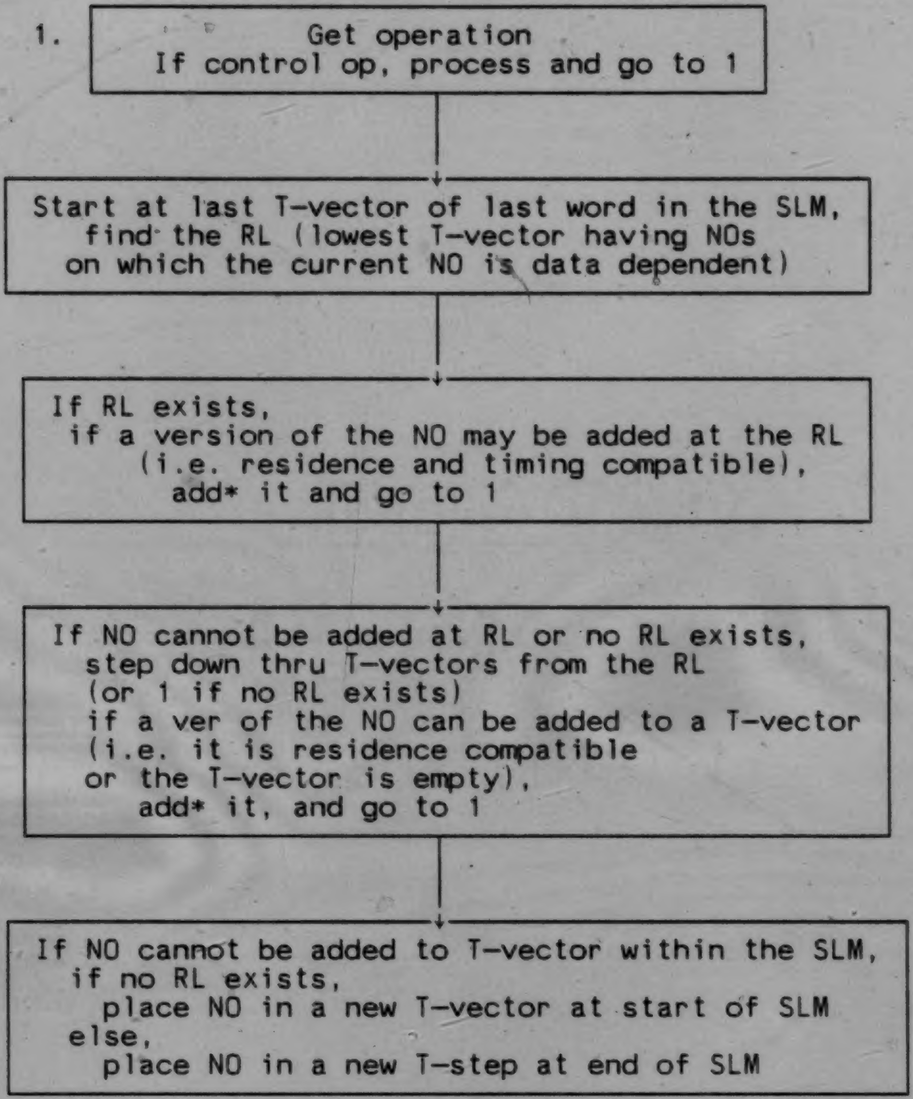
# APPENDIX I

## The LIN Algorithm

## General Algorithm


Read operations.
Save if a control op
Create an instance & save if an NO
Strip sequencing NOs from SLM
Bundle NOs in SLM on K-fields
Pack NOs into nanowords **
Assign nanostore locations to words
Enter sequencing NOs into words
Dump nanowords making up SLM
Fix up branch references and dump entire nanoprogram
For each SLM

## **Packing Phase



1. Get operation
If control op, process and go to 1

Start at last T-vector of last word in the SLM, find the RL (lowest T-vector having NOs on which the current NO is data dependent)

```
If RL exists,
 if a version of the NO may be added at the RL
     (i.e. residence and timing compatible),
        add* it and go to 1
```

```
If NO cannot be added at RL or no RL exists,
  step down thru T-vectors from the RL
  (or 1 if no RL exists)
  if a ver of the NO can be added to a T-vector
  (i.e. it is residence compatible
  or the T-vector is empty),
      add* it, and go to 1
```

```
If NO cannot be added to T-vector within the SLM,
  if no RL exists,
    place NO in a new T-vector at start of SLM
  else,
    place NO in a new T-step at end of SLM
```

* NO may be added if no bundling conflicts occur; if one occurs, perform a backup

# APPENDIX II

## Intermediate Language.

The intermediate language outlined here forms the means of interface between the S*(QM-1) compiler (translation phase) and the compactor. Input is in the form of triples, one to an input line, with the entries separated by blanks, tabs, or commas.

The first member of each triple identifies the nanooperation and is used as an input into a table holding the compactor's internal representation of the NO. This entry is shown under the *index* column in the tables that follow. The other two entries in a triple are parameters that give the compactor information for filling in related nanoword fields for the operation. These correspond loosely to the arguments included with the operations in the nanoassembler language.

The first argument is called the primary expression (*p expr*) and the other is the secondary expression (*s expr*). An unused parameter is identified by a value of -1. Only secondary expressions, where listed, may be left empty. All primary expressions must have an entry where one is listed in the table.

Most of the NO triples require no further explanation. Those for F-transfers (indices 45 through 49) do need some clarification of their use.

An input *aux* field may be specified as the particular field to be used, or the field may be left unspecified, and only the value to be transferred input. When the actual input *aux* field is to be specified (NO indices 45 and 47), one of the encodings listed on the last page of this Appendix is used to name that field. If it is the value to be transferred that is input, it is entered in the *p expr* field and the compactor chooses a K-field in which to place the value for transfer.

Destination *aux* fields must always be specified explicitly, and the encodings for these, which appear in the *s expr* field, are listed at the end of this Appendix. The *expr* in the operation list for these NOs refers to an F-register number.

The *stretch* NO never appears in the input stream and so is separated from the other NOs. Dummy NOs are generated by the compiler for resource usage delays.

The 'labels' input with some of the control operators are integers identifying the starts of SLMs that may be referenced by other SLMs.

*Input nanooperations:*

| **NO** | **index** | **p expr** | **s expr** |
|---|---|---|---|
| read ns | 0 | | |
| write ns | 1 | | |
| gate ns unc | 2 | | |
| gate ns () | 3 | test spec | val in test spec fld |
| skip () | 4 | test spec | val in test spec fld |
| read cs () | 5 | cs addr sel | |
| write cs () | 6 | cs addr sel | |
| gate cs | 7 | | |
| load es | 8 | | |
| gate es | 9 | | |
| gate alu | 10 | | |
| gate sh | 11 | | |
| load r31 | 12 | | |
| load npc () | 13 | lnpc | |
| inc mpc () | 14 | gspec | |
| index () | 15 | fsel2 | |
| in0 | 16 | | |
| in1 | 17 | | |
| in2 | 18 | | |
| out0 | 19 | | |
| out1 | 20 | | |
| out2 | 21 | | |
| msgo | 22 | | |
| msrs | 23 | | |
| read ms | 24 | | |
| gate ms () | 25 | rmi | |
| xio | 26 | | |
| rio | 27 | | |
| aux action | 28 | | |
| supervisor | 29 | | |
| generate int | 30 | gspec for T1 | |
| test= | 31 | test spec | |
| cs addr= | 32 | cs addr sel | |
| rmi= | 33 | rmi | |
| index reg ()= | 34 | aux2 | aux3 |
| f() | 35 | fsel1 | |
| gspec= | 36 | gspec | |
| carry control= | 37 | carry ctl | |
| aux0= | 38 | aux0 | |
| aux1= | 39 | aux1 | |
| aux2= | 40 | aux2 | |
| aux3= | 41 | aux3 | |
| fsel0= | 42 | fsel0 | |
| fsel1= | 43 | fsel1 | |
| fsel2= | 44 | fsel2 | |
| aux->expr (spec) | 45 | aux fld | f reg |
| aux->expr (unspec) | 46 | val for aux fld | f reg |
| aux->aux (spec) | 47 | aux fld | aux fld |
| aux->aux (unspec) | 48 | val for aux fld | aux fld |
| expr->aux | 49 | f reg | aux fld |
| legal micro op | 50 | | |
| allow nano int | 51 | | |
| allow micro int | 52 | | |
| allow interrupts | 53 | | |
| direct ms access | 54 | | |
| hold | 55 | | |
| hold2 | 56 | | |

| | | |
|---|---|---|
| shifter stat enable | 57 | |
| alu stat enable | 58 | |
| kn= | 59 | kn |
| branch () | 60 | kn |
| alt branch () | 61 | kn |
| prep branch () | 62 | kn |
| ka= | 63 | ka |
| kb= | 64 | kb |
| kalc= | 65 | kalc |
| kshc= | 66 | kshc |
| ksha= | 67 | ksha |
| kt= | 68 | kt |
| ks= | 69 | ks |
| kx= | 70 | kx |

*Used internally by compactor:*

| | |
|---|---|
| dummy | 71 |
| stretch | 72 |

*Control operators:*

| | | |
|---|---|---|
| slm start | −1 | |
| new nanoword | −2 | |
| new instruction | −3 | label |
| new subroutine | −4 | label |
| new interrupt | −5 | label |
| label | −6 | label |
| region start | −7 | |
| region end | −8 | |
| cocycle start | −9 | |
| cocycle end | −10 | |
| eof | −100 | |
| | | |
| reserved | −50 | |

*AUX field encodings for F-transfers.*

| **source aux field** | **index** |
|---|---|
| a | 0 |
| b | 1 |
| c | 2 |
| ka | 3 |
| kb | 4 |
| kx | 5 |
| kt | 6 |
| incf | 7 |
| decf | 8 |
| io id | 9 |
| aluf | 10 |
| g | 11 |
| sw | 12 |

| **destination aux field** | **index** |
|---|---|
| a | 0 |
| b | 1 |
| c | 2 |
| ka | 3 |
| kb | 4 |
| ks | 5 |
| kx | 6 |
| kt | 7 |
| kshc | 8 |
| kalc | 9 |
| ksha | 10 |

$CONTINUE WITH THES.SETUP RETURN

# APPENDIX III

## SLM Sequencing Conventions.

The compaction phase of the S*(QM-1) compilation process is responsible for generating and placing the nanooperations for sequencing between nanowords. Sequencing from word to word within an SLM is straightforward.

```
For instructions,
from first to second word of the SLM, generate
        load npc(kn);
        branch(lab), where lab=ns location of a word in tails area;
        read ns;
        gate ns unc;
from any other word to the next, generate
        load npc(seq);
        read ns;
        gate ns unc.
```

```
For subroutines and interrupt handlers,
from any word to the next, generate
        branch(lab), where lab=ns location of a word in tails area;
        read ns;
        gate ns unc.
```

Sequencing from the last nanoword of an SLM to the next is according to conventions established in conjunction with the design of the translation phase of the compiler. The operations placed in the last nanoword of the SLM are those passed to the compactor plus some complementing operations which are generated by the compactor. The following are the operations expected by the compactor for the various types of control transfers from the end of SLMs, and the operations generated by the compaction process.

**For instructions:**

1. *execute next instruction, and prefetch following instruction*
    - NOs provided by the compiler
        - load npc(cs)
        - load r31
        - read cs(*arg*) note 2
        - allow ints
    - NOs generated by the compactor
        - read ns
        - gate ns unc

2. *subroutine call*
    - NOs provided by the compiler
        - branch(*label*)
    - NOs generated by the compactor
        - load npc(seq)
        - read ns
        - gate ns unc

3. *'activate' statement (non-returning subroutine call)*
    - NOs provided by the compiler
        - branch(*label*) note 3
    - NOs generated by the compactor
        - read ns
        - gate ns unc

4. *'if' statement*
    - NOs provided by the compiler
        - alt branch(*label*)
        - gate ns(*cond*) note 4
    - NOs generated by the compactor
        - load npc(seq)
        - read ns
        - load npc(kn)
        - read ns
        - gate ns unc

5. *'goto' statement*
    - NOs provided by the compiler
        - branch(*label*)
        - load npc(kn)
    - NOs generated by the compactor
        - read ns
        - gate ns unc

6. *'if' statement with conditional operations* in a cocycle
    - NOs provided by the compiler
        - skip(*cond*) note 5
    - NOs generated by the compactor
        - load npc(seq) note 6
        - read ns
        - gate ns unc

7. *'repeat' statement*
    - NOs provided by the compiler

```
    gate ns(cond)                  note 7
NOs generated by the compactor
    branch(n+1)
    load npc(kn)
    read ns
```

**For subroutines and interrupt handlers:**

1. *unconditional return*
    NOs provided by the compiler
        gate ns unc
    NOs generated by the compactor
        read ns

2. *'if cond, return' statement (conditional return)*
    NOs provided by the compiler
        alt branch — note 8
        gate ns(*cond*)
    NOs generated by the compactor
        kn=n.+1
        read ns
        read ns
        gate ns unc

3. *execute next instruction, and prefetch following instruction*
    NOs provided by the compiler
        load npc(cs)
        load r31
        read cs(*arg*) — note 2
        allow ints
    NOs generated by the compactor
        read ns
        gate ns unc

4. *'goto' statement*
    NOs provided by the compiler
        branch(*label*)
    NOs generated by the compactor
        read ns
        gate ns unc

5. *'if' statement with conditional operations in a cocycle*
    NOs provided by the compiler
        skip(*cond*) — note 5
    NOs generated by the compactor
        branch(n.+1) — note 6
        read ns
        gate ns unc

6. *'repeat' statement*
    NOs provided by the compiler
        gate ns(*cond*) — note 7
    NOs generated by the compactor
        branch(n.+1)
        read ns

**Notes:**

1. Operations must appear in the order given.

2. May be placed in a cocycle with another operation, such as *inc mpc*.

3. The npc may subsequently be loaded with any value.

4. The first *read ns, gate ns* combination causes a sequencing to the conditionally executed statements; i.e. the branch is taken if the condition test succeeds.

5. The conditionally executed statements must all be in a cocycle. The *skip* operation cannot be placed in T-vector 3 of the nanoword.

6. The operations to sequence to the next word are generated only if the *skip* operation appears in T-vector 4 of the nanoword.

7. 'Repeat' statement bodies must fit into one nanoword. The *gate ns* operation is executed when the condition is satisfied and the following nanoword (after the 'repeat') is required.

8. The *alt branch* operation is passed to the compactor without a label. The destination of the branch is set by the compactor to be the next word. When executed, the first *gate ns* operation causes the return to the calling program.

# APPENDIX IV

## Compaction Examples.

The following are examples of the compactor performance in packing several routines from the QM-C instruction set [OLAF81], and from the MULTI instruction set [NANO76]. Each of these routines was hand-translated into the intermediate language given in Appendix II.

The QM-C instructions are first given in their source S*(QM-1) form, followed by the intermediate language translation (with the meaning of each triple), and finally the nanoassembler equivalent of the output of the compiler.

The MULTI instructions were translated from their nanoassembler form, and so their sources are listed in that manner. The intermediate language translation is included as for the QM-C routines, as well as the nanoassembler equivalent of the output.

Each example is followed by some comments on the time for execution of the compacted instruction, and how this compares with that obtainable by hand-nanoprogramming.

The QM-C routines use a number of macros that are expanded by the compiler. The listings of these macros are found after the last QM-C example, before the MULTI examples.

**QM-C CALL, Instruction Source**

```
proc p_call (instruction, mnemonic=call)

    /*                          Format 5  Size 18 bits
     *
     *  Procedure call.  First word of each procedure is a mask word indicating
     *                   the number of registers used (thus have to be saved)
     *                   and size of stack to be allocated for automatics and
     *                   temporaries.
     *
     *  Format   :  call  o11
     *
     *  Function : save registers 'pc' through 'reg[ ((eb)+o11)[ 17..11 ] ]' on stack
     *               (fp) <- (sp)
     *               (sp) <- (sp) - ((eb)+o11)[ 11..0 ]
     *               (pc) <- (eb)+o11+1
     *
     */


    INC_PROG_COUNT;                          /* Point pc to next instruction */
    scr1 := passl inst_reg;                  /* Save offset                  */


    mm_index_select := c_eb;                 /* Read from external base      */
    mm_addr_select := c_sp;                  /* Save on stack                */
    mm_data_select := c_pc_p1;               /* PC first saved (c_pc+1)      */


    MAIN_MEM_READ_I_L;                       /* Read mask word               */
    inst_reg := main_output;                 /* Prepare to decode            */



    repeat                                   /* Save registers               */
         mm_data_select := mm_data_select -1;
         main_memory[ mm_addr_so.reg[ mm_addr_select ] ] := mm_data_so[ mm_data_select ];


         ks := c_sp;
         reg[ gspec.ks ] := xdecl reg[ gspec.ks ];


         f_scr1 := mm_data_select - inst_reg.c
    until  (f_scr1 == 0);
    inst_reg.c := 0;                      /* Prep. to allocate automatics */
    fp := passl sp;                       /* Set new frame pointer        */
    sp := sp - inst_reg;                  /* Allocate for automatics      */

    alu_carry_in := 1;                    /* Add one to following expr.   */
    pc := eb + scr1;                      /* First instruction of proc    */

    mm_index_select := c_pc;
    MAIN_MEM_READ_I;                      /* Read onto bus for decode     */

    INSTRUCTION_FETCH_NEXT;               /* Activate instr. pipeline     */

endproc
```

*Intermediate Language Translation of CALL (QM-C)*

```
 -3 0 -1         new instruction(label 0)
 14 12 -1        mpc plus 1
 46 31 3         31->fail
 46 29 7         29->faod
 65 31 -1        kalc=pass left
 10 -1 -1        gate alu
 46 26 16        26->fmpc
 46 28 2         28->fcia
 46 26 4         26->fcid
 5 6 -1          read cs(mpc+ab)
 46 31 6         31->fcod
 7 -1 -1         gate cs

 -2 -1 -1        new nanoword
 45 8 4          decf->fcid
 6 0 -1          write cs(cia)
 -9 -1 -1        cocycle
 69 28 -1        ks=28
 36 14 -1        gspec='ks'
 34 5 -1         index reg('gspec')
 15 0 -1         index('decr left')
 -10 -1 -1       coend
 -9 -1 -1        cocycle
 44 4 -1         fsel2=fcid
 45 2 19         c->fiph
 41 2 -1         aux3='kt'
 68 6 -1         kt=sub
 45 10 20        aluf->g0
 -10 -1 -1       coend
 35 20 -1        f(g0)
 3 6 1           gate ns(x), kx=1

 -1 -1 -1        new sim
 49 19 2         fiph->c
 46 28 3         28->fail
 46 24 7         24->faod
 65 31 -1        kalc=pass left
 10 -1 -1        gate alu
 46 28 3         28->fail
 46 31 5         31->fair
 46 28 7         28->faod
 65 6 -1         kalc=sub
 10 -1 -1        gate alu
 37 2 -1         set cih
 46 26 3         26->fail
 46 29 5         29->fair
```

```
46 25 7         25->faod
65 9 -1         kalc=add
10 -1 -1        gate alu
46 25 16        25->fmpc
5 2 -1          read cs(mpc)
13 1 -1         load npc(cs)
12 -1 -1        load r31
5 3 -1          read cs(mpc+1)
53 -1 -1        allow ints
-100 -1 -1      eof
```

*Compactor Output*

```
0:
....  legal micro op, branch(220)
      ka=31, kb=29, kx=28, kt=26, kalc=pass left
S...  load npc(kn), kt->fmpc, kb->faod, ka->fail, mpc plus 1
.P..  read ns, ka->fcod, kt->fcid, kx->fcia, gate alu
..X.
...S  gate ns, gate cs, read cs(mpc+ab)

220:
....  branch(221), ks=28, kx=1, kt=6
S...  load npc(kn), decf->fcid
.P..  read ns, aluf->g0, fsel2=fcid, c->fiph, aux3='kt'
..S.  index('decr left'), index reg('gspec'), gspec='ks'
      write cs(cia), f(g0), gate ns(x)

221:
....  ka=28, kb=24, kx=31, kt=28, kalc=pass left
S...  load npc(seq), kb->faod, ka->fail, fiph->c
.P..  read ns, kt->faod, kx->fair, ka->fail, gate alu
..X.  gate ns

222:
....  ka=26, kb=29, kx=25, kt=25, kalc=sub
S...  load npc(seq), kt->faod, kb->fair, ka->fail, gate alu, set cih
.X..  read ns, kx->fmpc
..X.  gate ns

223:
....  allow ints, kalc=add
S...  gate alu
.X..
..S.  load npc(cs), read cs(mpc)
...S  gate ns, read cs(mpc+1), load r31, read ns
```

*Comments:*

The CALL Instruction consists of three SLMs, the second of which is the 'repeat' statement. The repeating word requires 7 T-periods per iteration (to save each register), and there is no apparent way to reduce this time (i.e. the optimal was obtained). The remaining words of the instruction require a total of 27 T-periods, of which 2 are caused by delays unnecessary within the SLM being placed in the first T-steps of the respective SLMs. Using F-transfers to dynamically load the *kalc* field would lead to some additional savings in the third SLM.

**QM-C ADDMM Instruction Source**

```
proc ADDMM (instruction)

    /*                           - Format 13 - Size 36 bits -
     *
     *   Add memory to memory. (medium short offsets)
     *
     *    Format   :  addmm    r1, o6, r2, o12
     *
     *    Function :  ((r2)+o12) <- ((r2)+o12) + ((r1)+o6)
     *
     */

    FETCH_OPRND_FORM_13;                         /* Fetch operands         */

    reg[ dest ] := reg[ alt_source ] + reg[ source ];
    SET_STATUS;

    act  write_reg;                              /* Write to memory, start pipe */

endproc
```

*Intermediate Language Translation for ADDMM (QM-C).*

```
 -3 0 -1          new instruction(label 0)
 14 12 -1         mpc plus 1
 45 0 3           a->fail
 49 19 0          fiph->a
 46 31 5          31->fair
 46 30 7          30->faod
 65 9 -1          kalc=add
 10 -1 -1         gate alu
 46 31 6          31->fcod
 7 -1 -1          gate cs
 46 30 2          30->fcia
 5 0 -1           read cs(cia)
 46 30 6          30->fcod
 7 -1 -1          gate cs
 45 2 3           c->fail
 49 19 2          fiph->c
 46 31 5          31->fair
 46 31 7          31->faod
 65 9 -1          kalc=add
 10 -1 -1         gate alu
 46 31 2          31->fcia
 5 0 -1           read cs(cia)
 46 29 6          29->fcod
 7 -1 -1          gate cs
 46 29 7          29->faod
 46 30 5          30->fair
 46 29 3          29->fail
 65 9 -1          kalc=add
 10 -1 -1         gate alu
 58 -1 -1         alu status enable
 57 -1 -1         sh status enable
 60 1 -1          branch(label 1)


 -4 1 -1          new subroutine(label 1)
 49 5 7           fair->kt
 45 6 4           kt->fcid
 6 0 -1           write cs(cia)
 5 3 -1           read cs(mpc+1)
 14 12 -1         mpc plus 1
 13 1 -1          load npc(cs)
 12 -1 -1         load r31
 5 3 -1           read cs(mpc+1)
 53 -1 -1         allow ints
 -100 -1 -1       eof
```

*Compactor Output*

```
0:
....  legal micro op, branch(220), sh status enable
      ka=31, kb=30, kalc=add
S...  load npc(kn), ka->fair, fiph->a, a->fail, mpc plus 1
.X..  kb->faod, kb->fcia, ka->fcod, read ns
..S.  gate cs, gate alu, fiph->c, c->fail, kb->fcod
...X  ka->fair, ka->faod, gate ns


220:
....  ka=31, kb=29
S...  load npc(seq), read cs(cia), gate cs, ka->fcia, kb->fcod
.S..  read ns, gate ns

221:
....  branch(222), ka=29, kb=30, kt=29, kalc=add
S...  load npc(seq), gate alu, kt->fail, kb->fair, ka->faod
.X..
..S.  read ns, read cs(cia), gate cs
...S  gate ns, gate alu

222:
....  branch(223)
X...  read ns, fair->kt
.X..  kt->fcid
..X.
...X  gate ns, write cs(cia)

223:
....  allow ints
S...  load npc(cs), read cs(mpc+1), mpc plus 1
.X..  load r31
..S.  read ns, gate ns, read cs(mpc+1)
```

*Comments:*

The ADDMM instruction consists of an instruction SLM and a subroutine SLM, each of which pack into two nanowords. The instruction SLM requires 14 T-periods to execute, one of which is extra on account of the 'unneeded delay' problem. Another T-period is extra as a result of the combination of the input order-dependence and the filling of empty T-steps by the compactor. The delay for F-register *fcia* (*fcia* is loaded in the first nanoword) is placed in the empty T-vector of that nanoword; if the F-transfer had been earlier in the SLM, or if the compactor could recognize that the *alu*-related operations would be forced to the next word, the fourth T-vector of the first nanoword would not have been used, saving a T-period.

There is no way to pack the subroutine to get a better time performance than that obtained by the compactor.

## QM-C PUSHM Instruction Source

```
proc PUSHM (instruction)

    /*                           - Format 3 - Size 18 bits -
     *
     *  Push variable to stack.
     *
     *    Format   :  pushm     r, o6
     *
     *    Function :  (sp) <- ((r)+o6), (sp) <- (sp)-1
     *
     */

    alt_source := inst_reg.ap;                   /* Calc. address of source  */
    inst_reg.ap := 0;
    scr1 := reg[ alt_source ] + inst_reg;


    addr := c_scr1;                              /* Get source value         */
    MAIN_MEM_READ_G;
    scr1 := main_output;


    mm_data_select := c_scr1;                    /* Set up for push          */
    mm_addr_select := c_sp;


    MAIN_MEM_PUSH_G;                             /* Push onto stack      */


    MAIN_MEM_READ_I;                             /* Reread next instruction */
    INSTRUCTION_FETCH_NEXT_P;                    /* Continue pipeline    */

endproc
```

*Intermediate Language Translation of PUSHM (QM-C)*

```
 -3 0 -1          new instruction(label 0)
 45 0 3           a->fail
 49 19 0          fiph->a
 46 31 5          31->fair
 46 29 7          29->faod
 65 9 -1          kalc=add
 10 -1 -1         gate alu
 46 29 2          29->fcia
 5 0 -1           read cs(cia)
 46 29 6          29->fcod
 7 -1 -1          gate cs
 46 29 4          29->fcid
 46 28 2          28->fcia
 6 0 -1           write cs(cia)
 -9 -1 -1         cocycle
 69 28 -1         ks=28
 36 14 -1         gspec='ks'
 34 5 -1          index reg('gspec')
 15 0 -1          index('decr left')
 -10 -1 -1        coend
 5 2 -1           read cs(mpc)
 14 12 -1         mpc plus 1
 13 1 -1          load npc(cs)
 12 -1 -1         load r31
 5 3 -1           read cs(mpc+1)
 53 -1 -1         allow ints
 -100 -1 -1       eof
```

*Compactor Output*

```
0:
....  legal micro op, branch(220)
      ka=31, kb=29, ks=28, kt=29, kalc=add
S...  load npc(kn), ka->fair, fiph->a, a->fail
.X..  read ns, kt->fcod, kb->fcia, kb->faod
..X.  kb->fcid, gate alu
...X  gate ns

220:
....  ka=28
S...  load npc(seq), ka->fcia, gate cs, read cs(cia)
.X..
..P.  index('decr left'), index reg('gspec'), gspec='ks'
      read ns, write cs(cia)
...S  gate ns, load npc(cs), read cs(mpc), mpc plus 1
```

```
221:
.....  allow ints
S...   load r31
.S..   gate ns, read ns, read cs(mpc+1)
```

*Comments:*

The PUSHM instruction consists of one SLM, packed to require 20 T-periods to execute. Only one extra T-period is apparent in the packing of this instruction, and that is due to the 'unneeded delays'.

## QM-C INCR Instruction Source

```
proc INCR (instruction)

    /*                              - Format 1 - Size 18 bits -
     *
     *  Increment register
     *
     *    Format    :  incr    r, c6
     *
     *    Function  :  (r) <- (r) + c6
     *
     */

    alt_source := inst_reg.ap;                       /* Set up to add const. */
    dest := inst_reg.ap;
    inst_reg.ap := 0;

    reg[ dest ] := reg[ alt_source ] + inst_reg;
    SET_STATUS;

    INSTRUCTION_FETCH_NEXT_P;                        /* Continue pipeline     */

endproc
```

*Intermediate Language Translation of INCR (QM-C)*

```
 -3 0 -1        new instruction(label 0)
45 0 7          a->faod
45 0 3          a->fail
49 19 0         fiph->a
46 31 5         31->fair
65 9 -1         kalc=add
10 -1 -1        gate alu
58 -1 -1        alu status enable
57 -1 -1        sh status enable
14 12 -1        mpc plus 1
13 1 -1         load npc(cs)
12 -1 -1        load r31
5 3 -1          read cs(mpc+1)
53 -1 -1        allow ints
 -100 -1 -1     eof
```

*Compactor Output*

```
0:
....  legal micro op, allow ints, sh status enable, alu status enable
      ka=31, kalc=add
S...  load npc(cs), mpc plus 1, fiph->a, a->fail, a->faod
.X..  load r31, ka->fair
..S.  gate ns, read ns, read cs(mpc+1), gate alu
```

*Comments:*

The INCR instruction is a comparatively short SLM, packed to take 5 T-periods in execution, one more than optimal because of the 'unneeded delays'.

## Subroutines and Macros Used by the QM-C Instructions

```
proc write_reg (subroutine)

    /*
     *  Write destination operand to memory and initiate instruction
     *  prefetch. Destination format : reg + offset.
     *
     *    Precondition                       Postcondition
     *      Value of destination            (main_memory[ (reg[ addr ]) ] = (reg[ dest ])
     *      in reg[ dest ].                 (main_memory[ (pc)+1 ] being decoded.
     *      Address of destination          (main_memory[ (pc)+2 ] being fetched.
     *      in reg[ addr ].
     *
     */

    mm_data_so := dest;                        /* Set up source for write */
    MAIN_MEM_WRITE_G;                          /* Write to memory  */


    MAIN_MEM_READ_P1;                          /* Read next instruction */
    INSTRUCTION_FETCH_NEXT_P;                  /* Start instruction pipeline */

endproc
```

```
macro INSTRUCTION_FETCH_NEXT_P

     /*
      *  Context of Activation :  At end of each instruction and internal
      *                           interrupt handlers.
      *
      *     As instruction i activates FETCH, decoding of instruction i+1
      *     is initiated concurrent with fetch of instruction i+2.
      *
      *     This macro also updates program counter.
      *
      */


     reg.index[ mm_index_select ] := reg.index[ cs_index_select ] +1;

     load_npc := c_mem;              /* Initiate decode/select next cycle  */
     load_r31 := 1;                  /* Load inst_reg from main_output bus */

     main_output := main_mem[ reg.index[ mm_index_select ] +1 ]

endmacro

macro INSTRUCTION_FETCH_NEXT

     /*
      *  Context of Activation :  At end of each instruction and internal
      *                           interrupt handlers.
      *
      *     As instruction i activates FETCH, decoding of instruction i+1
      *     is initiated concurrent with fetch of instruction i+2.
      *
      *     This macro assumes program counter already set.
      *
      */
```

```
    load_npc := c_mem;              /* Initiate decode/select next cycle  */
    load_r31 := 1;                  /* Load inst_reg from main_output bus */


    main_output := main_mem[ reg.index[ mm_index_select ] +1 ]


endmacro

macro  INC_PROG_COUNT

    /*
     *    Context of activation :  Instructions and interrupt handlers to
     *                             increment program counter by 1.
     *
     *              Assumes mm_index_select points to program counter.
     *
     */

    reg.index[ mm_index_select ] := reg.index[ mm_index_select ] +1

endmacro

macro  SET_STATUS

    /*
     *    Context of activation :  Arithmetic instructions and compare to
     *                             set global condition codes.
     *
     */

    alu_status   := 1;
    shift_status := 1

endmacro

macro MAIN_MEM_READ_I
```

```
        /*
         *  Read main memory    - Address from fp, eb or pc. "mm_index_select"
         *                        must be set up by caller to select any of these.
         *                        Data appears on the MM output bus.
         *
         */

        main_output := main_memory[ mm_addr_so.index[ mm_index_select ] ]

endmacro

macro MAIN_MEM_READ_P1

        /*
         *  Read main memory    - Address formed by adding one to
         *                        the index register (fp, ed or pc) selected by
         *                        "mm_index_select". Data appears on MM output bus.
         *
         */

        main_output := main_memory[ mm_addr_so.index[ mm_index_select ] +1 ]

endmacro

macro MAIN_MEM_READ_I_L

        /*
         *  Read main memory    - Address formed by adding the 11 bit "b" field
         *                        of the instruction register, sign extended to
         *                        the index register (fp, ed or pc) selected by
         *                        "mm_index_select". Data appears on MM output bus.
         *
         */

        main_output := main_memory[ mm_addr_so.index[ mm_index_select ] +ab ]

endmacro
```

```
macro MAIN_MEM_READ_G

        /*
         *  Read main memory     - Address from general register "mm_data_select"
         *                         must be set up by caller to select register.
         *                         Data appears on the MM output bus.
         *
         */

        main_output := main_memory[ mm_addr_so.reg[ mm_addr_select ] ]

endmacro

macro MAIN_MEM_WRITE_G

        /*
         *  Write main memory    - Address in general register. "mm_addr_select"
         *                         must be set up by caller to select any of these.
         *                         "mm_data_select" selects data source.
         *
         *     Note that data written appears on the MM output bus.
         *
         */

        main_memory[ mm_addr_so.reg[ mm_addr_select ] ] := mm_data_so[ mm_data_select ]

endmacro

macro MAIN_MEM_PUSH_G

        /*
         *  Write main memory    - Address in general register. "mm_addr_select"
         *                         must be set up by caller to select register.
         *                         "mm_data_select" selects data source.
         *                         Register decremented by 1 after write.
         *
         *     Note that data written appears on the MM output bus.
         *
         */
        main_memory[ mm_addr_so.reg[ mm_addr_select ] ] := mm_data_so[ mm_data_select ];


        ks := mm_addr_select;
        reg[ gspec.ks ] := xdecl reg[ gspec.ks ]

endmacro


macro FETCH_OPRND_FORM_13

   /*
    *  Get operands for instruction format 13           -       Size 36 bits.
    *
    *                    ((r2)+o12) <- ((r2)+o12)  op  (r1)+o6)
    *
    *  Precondition                                Postcondition :
    *    r1 in inst_reg.ap                           ((r2)+o12) in reg[ dest ]
    *    r2 on main_output[ 17..12 ]                    (r)+o12 in reg[ addr ]
    *    o12 on main_output[ 11..0 ]                  ((r1)+o6) in reg[ source ]
    *    o6  in inst_reg.bp
    *
    */


   INC_PROG_COUNT;                           /* Correct for 2-w instruction  */


   alt_source := inst_reg.ap;                /* Calc. address of source      */
   inst_reg.ap := 0;
   scr2 := reg[ alt_source ] + inst_reg;


   inst_reg := main_output;                  /* Prep. to decode dest. addr.  */


   addr := c_scr2;                           /* Get source operand           */
   MAIN_MEM_READ_G;
   scr2 := main_output;


   alt_source := inst_reg.c;                 /* Calc. address of destination */
   inst_reg.c := 0;
   inst_reg := reg[ alt_source ] + inst_reg;


   addr := c_inst_reg;                       /* Get destination operand      */
```

```
    MAIN_MEM_READ_G;
    scr1 := main_output;


    dest       := c_scr1;                   /* Set up for operation        */
    source     := c_scr2;
    alt_source := c_scr1

endmacro
```

## MULTI BZS Instruction Nanoassembler Source

```
BZS:
....   legal micro op, branch(b.relative), alternate, allow ints
S...   fist->ks, a->fist, load npc(cs)
.S..   read ns, gate ns(s)
       read cs(mpc+b), g('ks'), g->fist
..S.   read ns, gate ns, load r31, read cs(mpc+2), mpc plus 1


B.RELATIVE:
....   allow ints, ka=31
S...   load npc(cs), mpc plus b
.X..   read ns, ka->fair, ka->fcod
..X.   gate ns, load r31, read cs(mpc+1)
```

*Intermediate Language Translation of BZS (MULTI)*
(The instruction has been rearranged to conform to the compactor sequencing conventions).

```
-3 0 -1        new instruction(label 0)
-9 -1 -1       cocycle
45 0 18        a->fist
49 18 5        fist->ks
-10 -1 -1      coend
61 2 -1        alt branch(label 2)
3 2 -1         gate ns(s)


-1 -1 -1       new slm
-9 -1 -1       cocycle
36 14 -1       gspec='ks'
45 11 18       gspec->fist
-10 -1 -1      coend
13 1 -1        load npc(cs)
12 -1 -1       load r31
-9 -1 -1       cocycle
14 12 -1       mpc plus 1
5 4 -1         read cs(mpc+2)
-10 -1 -1      coend
53 -1 -1       allow ints


-4 2 -1        new subroutine(label 2)
5 5 -1         read cs(mpc+b)
14 14 -1       mpc plus b
46 31 5        31->fair
46 31 6        31->fcod
13 1 -1        load npc(cs)
12 -1 -1       load r31
5 3 -1         read cs(mpc+1)
53 -1 -1       allow ints
-100 -1 -1     eof
```

*Compactor Output*

```
0:
....  legal micro op, branch(220)
S...  load npc(kn), a->fist, fist->ks
.S..  gate ns, read ns

220:
```

```
....  alt branch(222)
X...  load npc(seq)
.S..  gate ns(s), read ns
..X.  load npc(kn)
...S  gate ns, read ns

221:
....  allow ints
S...  load npc(cs), g->fist, gspec='ks'
.S..  gate ns, read ns, read cs(mpc+2), mpc plus 1, load r31

222:
....  allow ints, ka=31
S...  ka->fcod, ka->fair
.S..  load npc(cs), mpc plus b, read cs(mpc+b)
..X.  load r31
...S  read ns, gate ns, read cs(mpc+1)
```

*Comments:*

The BZS Instruction was packed to require 14 T-periods if the branch is not taken (global status is non-zero), and 20 T-periods if the branch is taken. This compares to 9 and 13 T-periods respectively in the original MULTI version. The major source of difference is the necessity of rearranging the instruction to conform to the control transfer (sequencing) conventions expected by the compactor. Given the sequential input in that form, the compactor used only two extra T-periods over the hand-nanoprogrammed minimum.

## MULTI ADR Instruction Nanoassembler Source

```
ADR:
....   legal micro op, allow ints, alu status enable
       ka=31, kalc=add
S...   a->fail, b->fair, a->faod, clear cih, load npc(cs)
.X..   read ns, mpc plus 1, read cs(mpc+2)
..X.   gate alu, alu to coh, ka->fair, load r31, gate ns
```

*Intermediate Language Translation of ADR (MULTI)*

```
 -3 0 -1        new instruction(label 0)
 45 0 3         a->fail
 45 1 5         b->fair
 45 0 7         a->faod
 37 1 -1        clear cih
 65 9 -1        kalc=add
 10 -1 -1       gate alu
 58 -1 -1       alu status enable
 37 4 -1        alu to coh
 46 31 5        31->fair
 13 1 -1        load npc(cs)
 12 -1 -1       load r31
 -9 -1 -1       cocycle
 14 12 -1       mpc plus 1
 5 4 -1         read cs(mpc+2)
 -10 -1 -1      coend
 53 -1 -1       allow ints
 -100 -1 -1     eof
```

*Compactor Output*

```
0:
....   legal micro op, allow ints, alu status enable
       ka=31, kalc=add
S...   load npc(cs), clear cih, a->faod, b->fair, a->fail
.S..   alu to coh, gate alu, ka->fair
       read cs(mpc+2), mpc plus 1, load r31
       read ns, gate ns
```

*Comments:*

The ADR instruction was packed by the nanocode compactor to the optimal time. The difference in number of T-vectors used between the source and compactor-generated versions is immaterial.

## MULTI NGR Instruction Nanoassembler Source

```
NGR:
....   legal micro op, allow ints, alu status enable
       ka=31, kb=0, kalc=sub
X...   kb->faod, b->fair, ka->fail, set cih, load npc(cs)
.X..   faod->c, faod->b, a->faod->a
..P.   read ns, read cs(mpc+2), mpc plus 1
...X   gate ns, load r31, ka->fair, gate alu, alu to coh
```

*Intermediate Language Translation of NGR (MULTI)*

```
 -3 0 -1         new instruction(label 0)
 46 0 7          0->faod
 46 31 3         31->fail
 45 1 5          b->fair
 37 2 -1         set cih
 -9 -1 -1        cocycle
 49 7 2          faod->c
 49 7 1          faod->b
 45 0 7          a->faod
 49 7 0          faod->a
 -10 -1 -1       coend
 65 6 -1         kalc=sub
 10 -1 -1        gate alu
 58 -1 -1        alu status enable
 37 4 -1         alu to coh
 46 31 5         31->fair
 13 1 -1         load npc(cs)
 12 -1 -1        load r31
 -9 -1 -1        cocycle
 5 4 -1          read cs(mpc+2)
 14 12 -1        mpc plus 1
 -10 -1 -1       coend
 53 -1 -1        allow ints
 -100 -1 -1      eof
```

*Compactor Output*

```
0:
....  legal micro op, allow ints, alu status enable
      kb=31, kalc=sub
S...  load npc(cs), set cih, b->fair, kb->fail, ka->faod
.X..  alu to coh, faod->a, a->faod, faod->b, faod->c
      read ns, mpc plus 1, read cs(mpc+2), load r31
..X.  gate ns, kb->fair, gate alu
```

*Comments:*

The NGR instruction demonstrates the difficulty the compactor encounters with R31 and residual control. R31 is used as the left input to the ALU, and the *a* field of R31 is loaded after the residual control is set up. There should, therefore, exist a data dependence between the operations *faod->a* and *gate alu*, and a delay NO should be generated. This is not recognized by the compactor and so there is not enough time between the two operations in the output nanoword. The code produced is incorrect.

In this case the compactor can be changed to recognize the relationship between the NOs, since a constant is being placed in *fail*. However, if the value in the source field for that F-transfer is unknown to the compactor, this situation could go undetected.

**MULTI SLLI Instruction Nanoassembler Source**

```
SLLI:
....   legal micro op, allow ints, sh status enable
       kshc=single+left+logical
S...   a->fsid, a->fsod, b->ksha, load npc(cs)
.X..   read ns, read cs(mpc+2), mpc plus 1
..X.   gate ns, load r31, gate sh
```

*Intermediate Language Translation of SLLI (MULTI)*

```
 -3 0 -1        new instruction(label 0)
 45 0 8         a->fsid
 45 0 9         a->fsod
 47 1 10        b->ksha
 66 4 -1        kshc=single+left+logical
 11 -1 -1       gate sh
 57 -1 -1       sh status enable
 13 1 -1        load npc(cs)
 12 -1 -1       load r31
 -9 -1 -1       cocycle
 14 12 -1       mpc plus 1
 5 4 -1         read cs(mpc+2)
 -10 -1 -1      coend
 53 -1 -1       allow ints
 -100 -1 -1     eof
```

*Compactor Output*

```
0:
....   legal micro op, allow ints, sh status enable
       kshc=single+left+logical
S...   load npc(cs), b->fiph, fiph->ksha, a->fsod, a->fsid
.S..   gate sh, read cs(mpc+2), mpc plus 1, load r31
       read ns, gate ns
```

*Comments:*

The SLLI instruction was packed by the nanocode compactor to the optimal time. The difference in number of T-vectors used between the source and compactor-generated versions is immaterial.